175-WP-001-001

HDF-EOS Primer for Version 1 EOSDIS

White Paper

White Paper—Not intended for formal review or government approval.

April 1995

Prepared Under Contract NAS5-60000

RESPONSIBLE ENGINEER

Brand Fortner /s/	Doug Ilg /s/	4/10/95
Brand Fortner, Senior Analyst Doug Ilg, Senior Programmer Analyst EOSDIS Core System Project		April 10, 1995

SUBMITTED BY

Larry Klein /s/	4/10/95
Larry Klein, SDP Toolkit Manager EOSDIS Core System Project	April 10, 1995

Hughes Applied Information Systems
Landover, Maryland

This page intentionally left blank.

Abstract

This document is a preliminary release of a primer manual for using hierarchical data format (HDF) in Earth Observing System (EOS), Version 1 and later. We talk about how to use HDF today, and discuss plans for the future. In later releases of this document, we plan to explicitly describe how HDF files should be written by the EOS community. We hope to eventually supply an HDF-EOS subroutine library that will incorporate these HDF recommendations, and to supply tools that understand the conventions.

Note

This preliminary release contains many discussions for future implementations of EOS standards for using HDF. These discussions are just that: *discussions.* There are *no* definitive statements in this document, at least in this release. This document is being made available even in preliminary form because of the high level of interest in HDF-EOS efforts, and to give people the chance to comment on and to change our direction of HDF-EOS, long before decisions are burned into silicon, so to speak.

Credits

This document was created with material from Doug Ilg and R. Suresh (Hughes STX), Karen Whalen (Computer Sciences Corporation), Ted Meyer (National Aeronautics and Space Administration (NASA)), Larry Klein, Brand Fortner (Applied Research Corporation), and Mike Folk (National Center for Supercomputing Applications (NCSA)). Comments and suggestions should be sent to:

Brand Fortner
Applied Research Corporation
1616A McCormick Dr.
Landover, MD 20785
USA

Email: bfortner@eos.hitc.com
Phone: (301) 925-0779
Fax: (301) 925-0321

Keywords: EOS-HDF, Swath, Grid, Point, Data Formats, PVL, ODL, Standard Data Products, Disk Formats, Browse, Arrays

This page intentionally left blank.

Contents

3. The Future: HDF-EOS

Figures

Tables

Appendix A. Existing HDF Resources

This page intentionally left blank.

1. Introduction

1.1 Purpose of this Document

The purpose of this document is twofold. The first purpose is to introduce the Earth Observing System Data and Information System (EOSDIS) Version 1 (V1) community to the HDF file format that has been chosen as the EOSDIS Core System (ECS) Standard Data Format (SDF). Our intent is to provide enough background information so that EOS personnel that need to use HDF today can do so as easily as possible.

The second purpose of this document is to discuss our future plans for the EOS SDF. In particular, we talk about how we plan to create new types of structures that better map to EOS data, how we plan to add higher level support for these structures, and how we plan to provide tools to read and write these structures. Our intent is to provide a starting point for discussions on our future plans for the EOS SDF.

1.2 Organization of this Document

The first section concerns the present: an overview of the HDF file format, and how the current file format maps to EOS data. In particular, we go over basic HDF concepts such as general philosophy and core data types. We next discuss the current NCSA HDF library routines in detail.

The second section concerns the future: primarily our proposed HDF-EOS format. This new format will consist of the HDF standard with EOS conventions. We discuss the overall philosophy of HDF-EOS, the HDF-EOS data types that we will support, and issues such as browse data, metadata, and supplemental standards. In addition we talk about EOSView, our proposed 'HDF-EOS cracker tool.'

1.3 HDF and EOS: Introduction

The Hierarchical Data Format (HDF) has been selected by the EOSDIS Project as the format of choice for standard product distribution. HDF was originally developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois to help with the storage of supercomputer simulation results.

Documentation on the HDF disk format of HDF files is readily available. This is in contrast to network common data format (netCDF) and common data format (CDF), where the emphasis on the data model is so strong that documentation on the disk format does not exist.

At the most basic level, an individual HDF file consists of a directory and a collection of data objects. Every data object has a directory entry, containing a pointer to the data object location, and a flag defining the datatype for that data object. NCSA has defined around a hundred different datatypes; users can define additional datatypes.

Many of the NCSA defined datatypes map very well to proposed EOS datatypes. Examples would include raster images, multi-dimensional arrays, and text blocks. There are other EOS datatypes that do not map as well to NCSA datatypes. Examples would include grids, swath structures, and point data.

The HDF format is known for its generality, in that there are a very large number of legal ways to organize data in an HDF file. But this generality comes at a price: there is no guarantee that all data producers will store particular information such as geolocation in a particular way in the HDF files.

There are two ways to solve this problem: one way would be to explicitly define the layout of every EOS standard data product, and then incorporate these layout decisions in a subroutine library. The other method, and the one that we propose, is to define new EOS specific datatypes such as Grid, Swath, and Point, that contain information in a specific structure. That way, the HDF-EOS library has to know only about these structures, and not about every single data product.

In either case, the goal is to make HDF-EOS files completely self-describing, so no outside information will be needed to display the information contained in the files. The only caveat is that using the HDF-EOS library will make data access much more convenient for both producers and users.

1.4 HDF and EOS: General Philosophy

The following is a statement from the Earth Science Data and Information System (ESDIS) office concerning HDF:

> "The Earth Science Data and Information System Project has undertaken an analysis of available data format standards over the last 4 years. This analysis received input from Distributed Active Archive Centers (DAACs), EOS Instrument Investigators, related earth science projects, international investigators, computer scientists, and other members of the EOS community.
>
> As a result of this study, the ESDIS Project selected the National Center for Supercomputer Applications' Hierarchical Data Format (HDF) as the Standard Data Format (SDF) for Version 0 System distribution of science data.
>
> Based on successful experience in Version 0, including use by 6 DAACs, the Pathfinder project, and associated earth science projects, the ESDIS Project plans to adopt HDF as the baseline EOSDIS Standard Data Format for science and science-related data. On September 1993, HDF was adopted as a baseline standard for EOSDIS Core System development of standard data product generation, archival, ingest, and distribution capabilities.
>
> The ESDIS Project will support the evolution of the EOSDIS SDF as needed to meet the requirements of science data users and producers. "

Updated ESDIS statements concerning the use of HDF in the processing system will be added to this document at a later time.

2. The Present: HDF

The purpose of this part of the primer is to give a conceptual overview of HDF, and how EOS data may fit into the existing HDF structure. This material is *not* meant to be a replacement for the HDF manuals available from NCSA (see Appendix B), but instead to be a help in the comprehension of the concepts presented there.

2.1 HDF Concepts

HDF is a *disk format* and *subroutine library* for storage of most kinds of scientific data. HDF is intended for use in the storage of any kind of scientific data, although support is strongest for multi-dimensional arrays and raster images. It also contains very good support for the organization of data into hierarchical layers.

The *HDF disk format* is strictly binary, although ASCII text annotations are supported. There is a very strong emphasis on portability and machine independence, unusual for a binary format. A strength of HDF is that a single data file can contain several different *types* of objects. A color image of a molecule may be stored in the same file as the data object containing the actual positions of the atoms in space. The file may also contain an ASCII text annotation notebook describing the molecule.

At its most fundamental level, an HDF file consists of a directory and an unordered set of binary data elements. For the most part, the directory entries match up one-to-one with the binary data elements that follow. Each directory entry describes the location, the type, and the size of the corresponding element. We discuss the disk format in depth in Section 2.3.

The *HDF subroutine library* is designed to be very easy for C and FORTRAN programmers to use. Many simple HDF reads and writes can be accomplished with a single subroutine call. For example, to write a C character array that represents an 8-bit color image to an HDF file, the following HDF call is all that is required (the FORTRAN example is similar).

```
ret=DFR8addimage("myfile.hdf",image1,rows,cols,0);
```

Figure 2-1. HDF call to write an 8-bit image

This single call to the routine `DFR8addimage` creates the file '`myfile.hdf`', opens it, initializes '`myfile.hdf`' as an HDF data file, writes the image to '`myfile.hdf`', adds an HDF directory entry in the file for the image, and closes the file. The last argument in the call specifies the compression method, here `0` (no compression).

The HDF library is accessible from both C and FORTRAN programs because it contains a set of 'wrapper' functions that make the underlying C code callable from FORTRAN. Some FORTRAN compilers only accept function names that are eight or fewer characters. HDF therefore provides two names for each function; one for use in C programming and a shorter version for use in FORTRAN programming. For example, `d8aimg` is the FORTRAN equivalent for `DFR8addimage`. We discuss the HDF subroutine library in depth in Section 2.2.

Among widely used general scientific data formats, HDF may be unique in that the HDF libraries and manuals are in the *public domain*. This means that the HDF software and documentation can be used in commercial products without any licensing or even acknowledgment. The best source for HDF materials is via the NCSA anonymous file transfer protocol (ftp) server (See Appendix B).

2.2 Overview of the NCSA HDF Library

The HDF library can be thought of as three interface layers built upon a physical file format. The first interface layer, or the *low level layer*, is reserved for software developers. It provides supports for things like file I/O, error handling, memory management, and physical storage. It is essentially a software toolkit for skilled programmers who wish to make HDF do something more than what is currently available through the higher level interfaces.

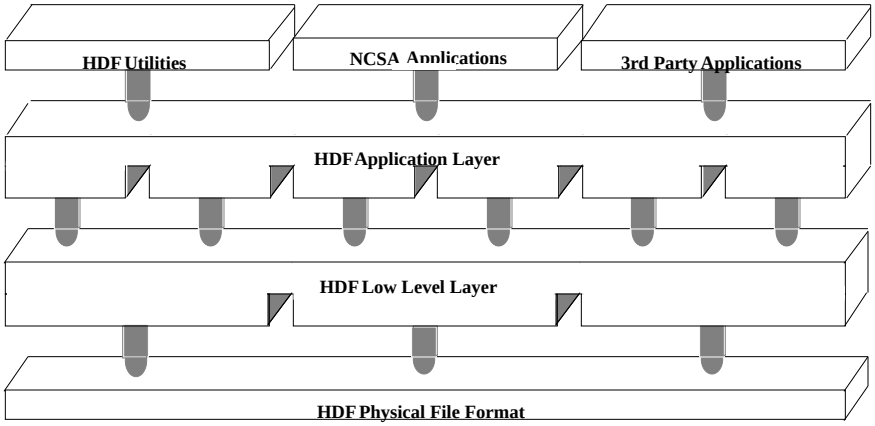

Figure 2-2. The HDF physical file format supports three levels of interaction.

Above the low level interface layer is the HDF *single file* and *multi-file application layer*. The application layer includes routines designed to simplify the process of storing and accessing data. Most HDF developers spend the majority of their time working with the application interface. Although each interface module requires some programming, all the low level details can be ignored.

At its highest level, HDF includes utilities, NCSA applications, and a variety of third party applications. Applications developed by NCSA, as well as applications contributed by HDF users, are freely available on the NCSA ftp server. In addition, several software vendors also support HDF.

The HDF library consists of callable routines in the low level layer and in the application layer. Underneath each layer, the routines are grouped into interfaces. Each interface addresses a particular HDF function or a particular HDF data structure. All the callable routines within a particular interface begin with the same letters. The different interfaces are therefore known by these letters. Table 2-1 below lists all the HDF interfaces, grouped by layer. Examples of callable routines from each interface are given in the last column of the table.

Table 2-1. The HDF Interfaces

Interface	Description	Example Routines (long names)
Low Level Layer Interfaces		
H	low level I/O, directory, query	Hopen, Hread, Hwrite, Hcreate
HDF	new version of low level routines	HDFopen, HDFclose
HE	low level error reporting	HEreport, HEprint
Single File Application Layer Interfaces		
DFR8	read, write 8-bit raster images	DFR8addimage, DFR8getdims
DFP	read, write color palettes	DFPaddpal, DFPgetpal
DF24	read, write 24-bit raster images	DF24addimage, DF24setdims
DFSD	single file scientific dataset	DFSDputdata, DFSDsetdimscale
DFAN	text annotation records	DFANputlabel, DFANgetdesc
Multi-file Application Layer Interfaces		
SD	multi-file scientific dataset	SDstart, SDcreate, SDdiminfo
NC	netCDF interface	nccreate, ncopen, ncvardef
VS	Vdata interface	VSattach, VSfdefine, VSgetid
VSQ	Vdata query	VSQuerycount, VSQueryname
VF	Vdata fields inquiry	VFfieldsize, VFfieldname
V	Access, Specify, Inquire Vgroups	Vattach, Vstart, Vsetname, Vgetid
VH	Simple Vdata, Vgroup creation	VHmakegroup, VHstoredata

The most important of these interfaces include the scientific dataset interfaces, the vdata interfaces, and the vgroup interfaces. They are described below.

2.2.1 The Scientific Dataset Interfaces (DFSD, SD)

There are two HDF interfaces that support multidimensional arrays. The older one is a single-file interface (known as the 'DFSD' interface, because all the associated subroutines start with 'DFSD') which permits access to only one file at a time. The newer one is a multi-file interface (known as the 'SD' interface), which permits simultaneous access to more than one file. We recommend the use of the newer multi-file interface for ECS users.

The single-file DFSD interface provides a collection of routines for reading and writing an array of arbitrary rank and type. The array, along with its associated information, is known as a *Scientific Data Set*, or SDS for short.

The multi-file SD interface allows concurrent operations on more than one file and data object. The interface is also interoperable with the netCDF interface. By interoperable, we mean the netCDF interface as implemented in HDF can read both netCDF files and HDF files. Like the single-file DFSD interface, a data object written with the multi-file SD interface includes the normal SDS data element (`DFTAG_SD`). However, the multi-file SD interface also includes many additional attributes that are not part of the single-file interface.

In either interface, the multi-dimensional array in the SDS can contain 8-, 16-, or 32-bit signed or unsigned integers or 32- or 64-bit floating point numbers. The SDS can also contain the following attributes:

- Coordinate system ___Identifies which coordinate system to use when displaying data.
- Formatting _____Specifies the format for displaying values for data and attributes.
- Label_____Contains a name for each independent variable and the data.
- Max/Min value _____Stores the maximum and minimum values in the data, as supplied by the data producer.
- Scale _____Describes the scale to use along each axis.
- Units _____Identifies the unit associated with each dimension and the data.

Section 2.3.5 of this document shows examples of SDS data objects in HDF files. The multi-file SD interface subroutines are divided into the following categories:

- Access routines that intialize and close the SD interface.
- Create, read, and write SDS routines for defining and reading array dimensions, rank, number type, fill value, data range, calibration information, and data values.
- Dimension attribute routines for defining and reading SDS attributes such as dimension name, format, unit, label, or scales. All attributes are optional.
- General attribute routines for managing local attributes (attributes assigned to a data object) and global attributes (attributes assigned to a file). Predefined local attributes include the coordinate system, format, labels, max/min values, scales, and units.

In EOS, we have given a strong emphasis to the multifile SD interface.

2.2.2 The Vdata Interfaces (VS, VSQ, VF)

The HDF Vdata model, which includes the VS, VSQ, and VF interfaces, makes it easy to store tables of data in HDF files. Each table consists of a series of records, each of which contains a series of fields. Each field can support its own number type. However, every record in a Vdata must contain the same fields. Valid number types include 8-, 16-, 32-bit signed and unsigned integers, 32- and 64-bit floating point numbers, and ASCII characters.

Vdata tables use three kinds of identifying information: a *name*, a *class*, and a set of individual field names. A Vdata name is a label typically describes the origin and contents of a table. A Vdata class typically identifies the meaning of data. Finally, Vdata field names identify the individual fields that make up a record.

Vdata Name	Temperature Table		
Vdata Class	Class_Table		
	Field #1	*Field #2*	*Field #3*
Field Names	Latitude	Longitude	Temperature
Record #1	8-bit Int	8-bit Int	32-bit Float
Record #2	8-bit Int	8-bit Int	32-bit Float
Record #3	8-bit Int	8-bit Int	32-bit Float

Figure 2-3. A Typical Vdata

There are three Vdata interfaces. The VS Interface provides a collection of routines for reading and writing tables. The VS functions are divided into four categories:

- Access routines which initialize and terminate access to a Vdata, and seek record position in a Vdata.
- Specify routines which define new Vdata fields, assign names and classes, and initialize read and write permissions.
- Inquiry routines which check if a Vdata exists, and return a Vdata's class name or field names, its size, and whether it exists as a lone entity.
- Read/Write routines which retrieve and store Vdata records in HDF files.

2.2.3 Vgroup interface (V)

The Vgroup interface provides a collection of routines for reading and writing groupings of HDF data objects in a particular HDF file. Each Vgroup may contain any number of other HDF data objects, even other Vgroups. In addition to its members, a Vgroup may also be given a Vgroup name and Vgroup class. Every function on Vgroup begins with the prefix V. The Vgroup functions are divided into four categories:

- Access routines that begin and end access to Vgroups.
- Specify routines which assign names, classes, and members to Vgroups.
- Inquiry routines which return the name and class of a Vgroup, and determine organization of the members of a Vgroup.

2.3 Overview of the HDF Disk Format

It is easier to understand the HDF software if you first have a general understanding of HDF disk formats. An HDF data file consists of three main things: a magic number, a directory that points to data elements, and the data elements themselves. We start by describing the HDF magic number.

2.3.1 The HDF Magic Number

The first four bytes of an HDF file must be `0E031301h` (where 'h' stands for hex), which is equal to the characters `^N^C^S^A` (where `^N` stands for control-`N`, etc.). This unique number identifies a file as an HDF file. The HDF library automatically writes this number to every file it creates and checks for the existence of the number in every file it opens.

The HDF documentation refers to this magic number as the 'file header'. The HDF definition of the term is in contrast to the more traditional view of a file header as a block of metadata at the beginning of a file. To avoid confusion over this point, we will not use the term 'file header' in this document. The directory begins immediately after the magic number.

2.3.2 The HDF Directory

An HDF directory, called a *DD list*, is usually broken up into a series of components known as *DD blocks.* Each DD block contains just three things: the number of directory entries (known as *Data Descriptors*, or *DD*s) in that block, the byte location of the next DD block (or 0 if this is the last DD block), and then the actual directory entries.

The first two bytes of each DD block contain a count of the number of entries in the block. The next four bytes contain the byte location of the next DD block. The actual DDs follow immediately. Every DD is always exactly 12 bytes long. The layout of an HDF file with a single DD block containing two DDs is shown below.

Table 2-2. Organization of the beginning of an HDF file

Location	Value	Comment
0 to 3	^N^C^S^A	Unique HDF Number (^N = control-N, etc.)
4 to 5	2	Number of DDs in DD Block
6 to 9	0000	Location of next DD Block (none here)
10 to 21	---	Data Descriptor #1
22 to 33	---	Data Descriptor #2

Note that the location of the next DD Block entry is given in *bytes from the beginning of the HDF file*. In fact, *all* locations within the HDF file are given in terms of bytes from the beginning of the file. This contrasts with some other disk formats which locate items solely by record numbers. There are no records, as such, in HDF. Since all locations are given in 32-bit *signed* integers, the maximum size of a self-contained HDF file or of a single data element is therefore around 2×10^9 bytes, or 2 GigaBytes.

2.3.3 HDF Data Descriptors

The single most important HDF concept is that of the Data Descriptors. *Every single data element (e.g., image, array, annotation, etc.) in the HDF file has an associated Data Descriptor (DD) in the DD list*. We will keep returning to this statement throughout the discussion.

Every DD is of fixed length with four fields: a *Tag* field, which defines the data element *type*, a *Reference Number* field (*Ref*), which is unique for every data element with the same Tag, an *Offset* field, which gives the location of the data element in bytes from the start of the file, and a *Length* field, which gives the length of the data element in bytes.

Table 2-3. HDF data descriptor layout

Data Descriptor											
1	2	3	4	5	6	7	8	9	10	11	12
Tag		Ref		Offset				Length			

2.3.3.1 HDF Tags

The Tag field is defined as a 16-bit unsigned integer, which means there are 65535 possible types of data elements. (0 is not a legal tag number.) The possible tag values are divided into three ranges, as shown below.

Table 2-4. Ranges of possible tag values

Tag Value Range	Comment
00001 to 32767	Assigned by NCSA
32768 to 64999	Defined by user
65000 to 65535	Reserved for future use

User-defined and NCSA-defined data element types can be freely intermixed in the same file. NCSA has defined *utility tags* for general data descriptions, *raster image* tags for descriptions of pseudocolor and color images, *scientific dataset* (SDS) tags for describing multi-dimensional arrays of numbers, the *Vdata* tag for defining tables of values, and the *Vgroup* tag for grouping data elements. A selection of these tags is shown below.

Table 2-5. A selection of NCSA defined HDF tag types

Tag Name	Tag #	Comments
	Utility Tags	
DFTAG_RLE	011	Specifies the Run Length Encoding used for image
DFTAG_TID	102	Tag Identifier: text string of user defined tag
DFTAG_DIL	104	Data Identifier Label: used for titles of elements
DFTAG_DIA	105	Data ID Annotation: lengthy text annotation block
DFTAG_NT	106	Number Type: values are float, integer, text, etc.
DFTAG_MT	107	Machine Type: specifies IEEE or local computer type
	Raster Image Tags	
DFTAG_ID	300	Image Dimension: gives X and Y size of assoc. image
DFTAG_LUT	301	Lookup Table: color lookup table for assoc. image
DFTAG_RI	302	Raster Image: points to actual image data
DFTAG_RIG	306	Raster Image Group: lists all DDs assoc. with image
DFTAG_LD	307	LUT Dimension: size of color lookup table
DFTAG_CFM	311	Color Format: grayscale, Pseudocolor, RGB, HSI, etc.
	Scientific Dataset Tags	
DFTAG_SDD	701	SDS Dimension: dimension sizes of scientific dataset
DFTAG_SD	702	Scientific Data: points to actual scientific dataset
DFTAG_SDS	703	Scientific Data Scales: Arrays for X,Y,Z locations
DFTAG_SDL	704	SD Labels: text describing data and dimensions
DFTAG_SDU	705	SD Units: text with units for data and dimensions
DFTAG_SDF	706	SD Format: text with format code for displaying data
DFTAG_SDM	707	SD Max/Min: minimum/maximum valid values for data
DFTAG_SDC	708	SD Coordinate System: text string defining CS
DFTAG_NDG	720	Numeric Data Group: lists all DDs assoc. with SD
	Vgroup/Vdata Tags	
DFTAG_VG	1965	Vgroup: provides general purpose grouping of DDs
DFTAG_VH	1962	Defines the structure of a Vdata data element
DFTAG_VS	1963	Points to Vdata data element using DFTAG_VH format

2.3.3.2 HDF Reference Numbers

If you store two raster images in an HDF file, you will produce two DDs with the same tag number (`DFTAG_RI`). But this creates a problem: how does one know which `DFTAG_RI` is which?

The answer is that a *Reference number* is assigned, usually by the HDF library, to each data object as it is written to the file. The library keeps track of the reference numbers that have been used in the file and guarantees that there will never be two data objects with the same tag *and*

reference number in the same file. It *is* allowable for two objects to carry the same tag (such as our two raster images) or for two objects with different tags to carry the same reference number (such as a raster image and an SDS).

Although many DDs can have the same Reference number, the HDF standard demands that there is only one instance of a particular Reference number for a particular *Tag*. This means that every *Tag/Ref* combination is *unique* within a single HDF file. A *Tag/Ref* pair can then be used as a unique ID (known in HDF terms as a '*data identifier*') to unambiguously identify particular data elements.

It is important to note at this point that, with only one exception that we will discuss in a later section, one must take care not to infer meaning on the library's choice of reference numbers. It will often be the case that some number of data objects in a file will have the same reference number. The only information that can be gleaned from this similarity in reference numbers is that the library is miserly in doling out new reference numbers. It reuses reference numbers in all cases where it can easily ascertain it to be safe to do so. Coincidentally, such cases often occur when writing out groups of data objects such as Raster Image Groups and Scientific Data Sets. However, this behavior is not an official part of the specification of HDF and it should never be relied upon.

Although it would be tempting to use Reference numbers to group data elements, that method of associating data objects is lacking in that it does not support sharing of data objects between groups. Instead HDF uses *Groups* to define relationships.

2.3.4 HDF Groups

The HDF libraries support explicit grouping of data elements using one of three group tags. The Raster Image Group Tag (`DFTAG_RIG`) is used to group all data objects associated with a particular *raster image*. The Numeric Data Group Tag (`DFTAG_NDG`) is used to group all data objects associated with a particular *scientific dataset.* The Vgroup tag (`DFTAG_VG`) supports a generalized grouping of *all* types of data objects, even other Vgroups.

Table 2-6. HDF group tags

Tag Name	Tag #	Comments
DFTAG_RIG	306	Raster Image Group: lists all DDs assoc. with image
DFTAG_NDG	720	Numeric Data Group: lists all DDs assoc. with SDS
DFTAG_VG	1965	Vgroup: provides general purpose grouping of DDs

The data contained within these groups is a list of *Data Identifiers* associated with the group. Since the groups do *not* contain the offset values of the data elements, a DD is *still* needed in the DD list with this information. So again, *all* data elements must have a DD in the DD list. In addition, *most* will *also* have their *Data Identifiers* listed in some group.

Both *Raster Image Groups* (*RIGs) and Numeric Data Groups (NDGs)* consist of nothing but a list of *Data Identifiers*, each four bytes long (*Vgroups* are more complicated; see below). The number of *Data Identifiers* in a RIG or NDG is just the length field divided by four.

In the table below, we show an HDF example file that uses Raster Image Groups (RIGs) to define two images of identical size. Note how, in this example, both raster image groups use the same image size data element.

Table 2-7. Organization of an HDF file with two raster image groups

Location	Length	Value				Comment
0 to 3	4	^N^C^S^A				Unique HDF Number
4 to 5	2	5				Number of DDs in Block
6 to 9	4	0000				Loc. of next DD Block
		Tag	*Ref*	*Offset*	*Length*	
10 to 21	12	300	001	70	4	DD #1 (Image Size)
22 to 33	12	302	001	90	60000	DD #2 (Ptr to Image#1)
34 to 45	12	302	002	60090	60000	DD #3 (Ptr to Image#2)
46 to 57	12	306	005	74	8	Ptr to 1st raster group
58 to 69	12	306	006	82	8	Ptr to 2nd raster group
		X Size		*Y Size*		
70 to 73	4	300		200		Size of both images
		Tag/Ref#1		*Tag/Ref#2*		
74 to 81	8	300/001		302/001		First raster group
82 to 89	8	300/001		302/002		Second raster group
Image Data						
90 to 60089	60,000	02h……23h				Data for first image
60090 to 120089	60,000	7Ah……19h				Data for second Image

RIGs can contain only raster image and related tags, and all the *Data Identifiers* in a particular group must relate to a single image. The same holds true for an NDG; only numeric data group tags (and some utility tags such as number type) are allowed, and they must all relate to a single numeric array.

Vgroups do not have this limitation: they can contain any collection of *Data Identifiers*, including a *Vgroup* Data Identifier. This last feature means that you can construct a directory structure inside a single HDF data file, with directories containing data elements and/or sub directories.

There is, however, a major difference between Vgroup organization and disk directories. In an HDF file, all groups including Vgroups contain *Data Identifiers only*, not full DDs. This means that a DD must still be in the main DD list for *every single data element* in the HDF file, *regardless* of how deeply it is referenced in a Vgroup hierarchy.

For example, consider the HDF file shown below. (We have cheated a bit in this table, because the structure of Vgroup data elements is a bit more complicated than what is shown.)

Table 2-8. Organization of an HDF file with two Vgroups

Location	Length	Value				Comment
0 to 3	4	^N^C^S^A				Unique HDF number
4 to 5	2	3				Number of DDs in block
6 to 9	4	0000				Loc. of next DD block
		Tag	*Ref*	*Offset*	*Lth*	
10 to 21	12	300	001	46	4	DD #1 (Image size)
22 to 33	12	1965	001	50	22	DD #2 (Vgroup #1)
34 to 45	12	1965	002	72	22	DD #3 (Vgroup #2)
		X Size		*Y Size*		
46 to 49	4	300		200		Image size
		Name		*Tag/Ref*		
50 to 71	22	'Vone'		1965/002		First VGroup
72 to 93	22	'Vtwo'		300/001		Second VGroup

How this file is organized may not be immediately obvious. The graphic below may help. Here we show that the main DD list contains the '`Vone`' Vgroup, which contains a single *Data Identifier*, a reference to the '`Vtwo`' Vgroup. This Vgroup in turn contains a single *Data Identifier* reference to the Image Size entry.

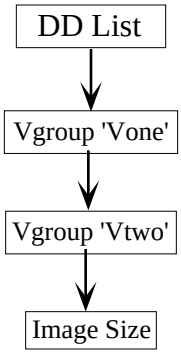

Figure 2-4. Organizational Levels for Dataset Shown in Table 2-8

The confusion is that every *Data Identifier*, even the one for '`Vtwo`', is *also* listed in the DD list. How does someone know which Vgroup is at what level? The answer is that there is no unique way.

You can use the HDF utility 'vshow' to display a list of the contents of each directory. But what should the utility do if '`Vone`' contained '`Vtwo`', *and* '`Vtwo`' contained '`Vone`'? Can a directory contain a directory that contains the original directory? Again, it is the responsibility of the data producer to ensure that the directory structure makes sense.

2.3.5 HDF Scientific Datasets

The most important data objects in HDF files are known as *Scientific Datasets* (SDS). A scientific dataset is a multidimensional array of numbers. The array can have any dimensionality up to 32767, and can consist of floating point or integer values. Since the release of HDF version 3.3 there have been two separate interfaces for SDSs: the 'DFSD' interface (old) and the 'SD' interface (new). Each interface corresponds to a different physical organization of the SDS and, therefore, has slightly different capabilities. The most significant new features of the 'SD' interface are its ability to concurrently deal with multiple SDSs in multiple files, its ability to put arbitrary attributes on SDSs, and its compatibility with the netCDF interface.

Although the 'SD' interface is the preferred interface for reading and writing SDSs in ECS, its physical implementation is a bit too complex to describe here. Therefore, we will discuss the physical implementation of the older 'DFSD' interface in order to illustrate the basic organizational concepts of the HDF library, then briefly touch on the differences between the two implementations.

2.3.5.1 The 'DFSD' Interface

An SDS, whether created by the 'DFSD' or 'SD' interface, consists of several data objects that are associated with the SDS. These data objects are attributes, numerical scales, the actual data, and so on. In the 'DFSD' interface, all the DDs of the data objects associated with the a particular SDS are listed in a *numeric data group* (`DFTAG_NDG`). In some old HDF files with 32-bit floating point SDSs, you may run across an SDS that uses the roughly equivalent *scientific data group* (`DFTAG_SDG`) to collect its members. In either case, only scientific data set tags (Table 2-5) and a few utility tags are allowed in a numeric data group. An example of the data contained in an SDS is shown below.

Neutron Star Accretion Simulation

Radius (km) \ Time (sec)	1.67	1.68	1.69	1.70
32.11	0.5872	0.5872	0.5872	0.5872
31.96	0.5872	0.5872	0.5872	0.5872
31.81	0.5872	0.5872	0.5872	0.5872
31.66	2.4226	2.3604	2.4282	2.4117
31.51	1.9976	1.9957	1.9963	2.0018
31.36	1.8119	1.8102	1.8107	1.8153
31.20	1.6768	1.6726	1.6745	1.6796
31.05	1.5762	1.5736	1.5742	1.5802
30.90	1.4981	1.4942	1.4956	1.4997
30.75	1.4356	1.4319	1.4319	1.4352

Density (gm/cm^3)

Figure 2-5. Example SDS

The data identifiers stored in the NDG for this SDS are shown below, along with the information pointed to by the data identifiers. In this list we ignore the actual format of the data identifiers, and concentrate instead on the information contained that they point to. Note also that the *Ref*, *Offset*, and *Length* fields for each DD are not shown.

Table 2-9. Data Objects for the SDS shown in Figure 2-4

Tag Name	Value(s)			Comments
	Data	*Dim1*	*Dim2*	
DFTAG_DIL	'Neutron Star Accretion Simulation'			Title for SDS Array
DFTAG_SDD	2 Dimensions			SD Dimension record
		4	10	Size of dimensions
	Float32	Float32	Float32	Number types
DFTAG_SDS		-1.67…1.70	30.75…32.11	SD Numerical scales
DFTAG_SDL	'Density'	'Time'	'Radius'	SD Labels
DFTAG_SDU	'gm/cm^3'	'sec'	'km'	SD Units
DFTAG_SDF	F7.4	F5.2	F5.2	SD Numerical Formats
DFTAG_SD	'0.5872,0.5872,……,1.4319,1.4352'			Actual data

The `DFTAG_SDD` tag says that the data set consists of two dimensions, sized four and ten elements, respectively, where the data and both scales consist of floating-point numbers. The numerical scales are set by `DFTAG_SDS`, with a four-element array for the first dimension and a ten-element array for the second dimension. As mentioned before, all of these data identifier entries are listed both in the main DD list directory, and in the NDG that defines the SDS.

2.3.5.2 The 'SD' Interface

The basic ideas behind the SD version of the SDS are the same as those in the DFSD SDS, but some changes were needed in order to achieve the major design goal: compatibility with netCDF. This driver gave rise to three distinct differences between the two implementations.

First, rather than using a specialized tag such as `DFTAG_NDG` to group its elements, the SD interface uses a general purpose Vgroup. Using a Vgroup allows for more flexibility in the organization of the SDS without a complete redesign. Second, the handling of dimensions has changed considerably, in that they can now have their own attributes. Third, the SD interface allows the use of user-defined attributes, not just the ones that are pre-defined.

Below is a list of the different styles of SDS, in order of their adoption into the HDF library.

Scientific Data Groups (SDG)—Originally, all SDSs were written in SDG groups. An SDG group (tag value = `DFTAG_SDG`) can contain any of the tags that are currently used for NDGs. The only difference between SDGs and NDGs is that SDGs only support 32-bit floating point arrays, whereas NDGs can support several types of floating point and integer number types.

SDGs are considered obsolete, although the HDF libraries will write out *both* an SDG *and* a NDG for the same array, if that array is made up of 32-bit floating point values. Application programs that are linked with versions of HDF before 3.2 will only recognize SDGs as SDSs. We will not discuss SDGs further, since they will not be used in ECS.

Numeric Data Groups (NDGs)—What we have just described in detail is the disk format for NDG groups (tag value = `DFTAG_NDG`). The NDG tag was new with HDF version 3.2. Application programs linked with version 3.2 of HDF will recognize SDGs and NDGs as SDSs. They are referred to in the latest documentation (3.3) as the "Old Style" SDS, although they are not *that* old. HDF subroutines that deal solely with NDGs and SDGs are prefixed with "`DFSD`".

Multifile SDSs—The recommended interface for SDSs in the current version of HDF (3.3) is known as the "Multifile SDS" interface, or the '`SD`' interface (after the prefix of the subroutine calls). The Multifile SDS interface defines a set of conventions and a set of library subroutines that uses Vgroups, Vdatas, and many of the parts of the NDG to create structures that are compatible with netCDF data structures.

The package is called Multifile SDS because the new subroutines allow several HDF files to remain open simultaneously, something not possible with the earlier libraries. Another significant advantage of the interface is the ability to assign arbitrary attributes to SDSs. In the documentation they are also referred to as "New Style" SDSs, or (confusingly) just SDSs. The multifile interface will recognize all older forms of the SDS.

We plan to use the Multifile SDS interface almost exclusively for EOS data, and we recommend that current EOS data producers use this interface where possible.

2.3.6 HDF Vdatas

HDF also supports the storage of tables that are organized as named columns known as *Vdatas*. Storage of a Vdata table requires the use of two tags: `DFTAG_VH` for defining and naming the columns of values; and `DFTAG_VS` for pointing to the actual Vdata data itself. There is no explicit grouping needed for Vdatas, although they can be included in a Vgroup (and usually are).

For example, consider the table of values shown below:

Table 2-10. Example Table of Values

ID. No	Flux	Name
1	2.34	CygX1
2	-89.43	HerX1
3	0.0023	CygX3
4	1.115	Vela

The tags needed to define this table as a Vdata data object are shown below. Again, we focus on the information pointed to by the tags and ignore the actual binary formats. In particular, we have assigned names to the various fields defined in the data record pointed to by `DFTAG_VH`. These field names are also used in the HDF documentation.

Table 2-11. Vdata tags for dataset shown in Table 2-10

Tag Name	Field Name	Values			Comments
		Col1	*Col2*	*Col3*	
DFTAG_VH	<nvert>	4			Number of records
	<isize>	11			Row width in bytes
	<nfields>	3			Number of fields
	<type>	int16	float32	char[5]	Field number types
	<isize>	2	4	5	Field size in bytes
	<offset>	0	2	6	Byte offset of field
	<fldnmlen>	5	4	4	Length of field Name
	<fldnm>	'ID.No'	'Flux'	'Name'	Field names
	<namelen>	19			Length of Vdata name
	<name>	'Vdata Example Table'			Vdata name
DFTAG_VS		1,2.34,'CygX1',…,'Vela'			Actual data

In this example, the information in the `DFTAG_VH` data element defines a Vdata with three *fields* (columns) and four Vdata *records* (rows). The values in these fields are defined as short (2-byte) integer, 4-byte floating-point, and five-character ASCII text, respectively. In addition, each field

has an ASCII text name. Note, by the way, how the entire Vdata definition can be named (<name>). This is unusual, in that most HDF data objects require a separate tag (DFTAG_DIL) to get a name.

The actual data pointed to by the DFTAG_VS tag is stored, packed as tightly as possible. The total width of every record is therefore 2 + 4 + 5 = 11 bytes. The four records of this table take up 11 × 4 = 44 bytes. This packing is shown below. Here the DFTAG_VS record associates itself with a DFTAG_VH record by having the same *Ref* number. This is the only case where the HDF libraries use the *Ref* numbers explicitly to associate two data elements.

Table 2-12. Disk layout of Vdata described in Table 2-11

	Byte Position										
	1	2	3	4	5	6	7	8	9	10	11
	Integer		*Floating Point*				*Text String*				
Record #0	1		2.34				C	y	g	X	1
Record #1	2		-89.43				H	e	r	X	1
Record #2	3		0.0023				C	y	g	X	3
Record #3	4		1.115				V	e	l	a	

2.3.7 HDF Extended Tags

The organizational features of HDF are so powerful that it is usually possible to store a complete data product granule in a single HDF file. There are, however, a couple of problems with very large single file data products.

The first problem, which we have already mentioned, is that HDF files are limited to 2 gigabytes in size. The second problem is that HDF data elements were not initially designed to be appendable. Normally, data elements are required to be in contiguous storage and they are packed right next to each other. Therefore, to make a data element bigger, it needs to be copied to the end of the file where there is room to grow, leaving a gap in the file.

To get around these limitations, NCSA has defined *extended tags*. An extended tag allows a data element to be spread among multiple locations in the HDF file or even in a completely separate file. A data element corresponding to any NCSA defined tag value can be converted to an extended tag, although this functionality is currently only supplied for SDSs and Vdatas.

An extended tag DD does *not* point directly to the data, as the normal tags do. It instead points to a data object *defining where the data is and how it is stored*. This data object *may* point to the beginning of a linked list of data blocks that contain the entire data record. This way, a data record can be lengthened just by adding an additional data block; the entire HDF file does *not* need to be rewritten.

Alternatively, the extended tag record could define the data element as being stored in an *external element* in another disk file. This feature (which is also found in CDF), allows a user to get around the two-gigabyte limitation on total file size. It may also make a large HDF 'file' easier to handle. Several 100-megabyte files are sometimes easier to handle than a single gigabyte file.

2.4 HDF Examples

The following sections contain sample C code for writing and reading a selection of basic HDF data objects. For now, we have kept to some very simple code that, for the most part, has been copied from NCSA's HDF documentation.

2.4.1 8-bit Raster Image Output

The C program below writes an 8-bit raster image to the file "example.hdf". This program takes no input.

```
#include "hdf.h"
#define WIDTH    5
#define HEIGHT    6

main(int argc, char *argv[])
{
    /*  Initialize the image array  */
    static uint8 raster_data[HEIGHT][WIDTH] =
                { 1,  2,  3,  4,  5,
                  6,  7,  8,  9, 10,
                 11, 12, 13, 14, 15,
                 16, 17, 18, 19, 20,
                 21, 22, 23, 24, 25,
                 26, 27, 28, 29, 30 };

    /*  Write the 8-bit raster image to the file  */
    DFR8addimage("example.hdf", raster_data, WIDTH, HEIGHT, 0);
}
```

2.4.2 8-bit Raster Image with Palette Output

The program below writes an 8-bit raster image and its associated palette to the file "example.hdf". This program takes no input.

```
#include "hdf.h"
#define WIDTH    5
#define HEIGHT    6

main(int argc, char *argv[])
{
    uint8 palette_data[768];
    intn i;

    /*  Initialize the image array  */
    static uint8 raster_data[HEIGHT][WIDTH] =
                { 1,  2,  3,  4,  5,
                  6,  7,  8,  9, 10,
                 11, 12, 13, 14, 15,
```

```
                16, 17, 18, 19, 20,
                21, 22, 23, 24, 25,
                26, 27, 28, 29, 30 };

    /*  Initialize the palette to standard linear grayscale  */
    for (i=0; i<256; i++) {
    palette_data[i*3] = i;
    palette_data[i*3+1] = i;
    palette_data[i*3+2] = i;
    }

    /*  Associate the palette with the image  */
    DFR8setpalette(palette_data);

    /*  Write the 8-bit raster image to the file  */
    DFR8addimage("example.hdf", raster_data, WIDTH, HEIGHT, 0);
}
```

2.4.3 8-bit Raster Image with Palette Input

The program below reads an 8-bit raster image and its associated palette from the file "example.hdf" created by the program in the section above. This program produces no output.

```
#include "hdf.h"
#define WIDTH    5
#define HEIGHT    6

main(int argc, char *argv[])
{
    uint8 raster_data[HEIGHT][WIDTH], palette_data[768];
    intn haspal;
    int32 width, height;

    /*  Get dimensions and check for palette  */
    DFR8getdims("example.hdf", &width, &height, &haspal);

    /*  Read the 8-bit raster image and palette from the file  */
    if ((width == WIDTH) && (height == HEIGHT) && (haspal == 1))
        DFR8getimage("example.hdf", (uint8 *) raster_data, width,
                     height, palette_data);
}
```

2.4.4 Scientific Data Set Output (Obsolete)

The code in this section makes use of the old SDS interface that was available prior to version 3.3 of the HDF library. We recommend that people use the new interface, shown in section 2.4.6 and 2.4.7. This section is included primarily for pedagogical reasons.

The program below writes a Scientific Data Set to the file "example.hdf". This program takes no input.

```
#include "hdf.h"
#define LENGTH    3
#define HEIGHT    2
#define WIDTH     5

main(int argc, char *argv[])
{
    /*  Initialize the image array  */
```

```
    static float64 scien_data[LENGTH][HEIGHT][WIDTH] =
                { 1.,  2.,  3.,  4.,  5.,
                  6.,  7.,  8.,  9., 10.,
                 11., 12., 13., 14., 15.,
                 16., 17., 18., 19., 20.,
                 21., 22., 23., 24., 25.,
                 26., 27., 28., 29., 30. };
    int32 dims[3] = {LENGTH, HEIGHT, WIDTH};

    /*  Set number type to 64-bit float  */
    DFSDsetNT(DFNT_FLOAT64);

    /*  Write the data array to the file  */
    DFSDadddata("example.hdf", 3, dims, scien_data);
}
```

2.4.5 Scientific Data Set Input (Obsolete)

The program below reads an old-style Scientific Data Set from the file “example.hdf” created by the program in section 2.4.4. This program produces no output.

```
#include "hdf.h"
#define MAXRANK  3
#define LENGTH   3
#define HEIGHT   2
#define WIDTH    5

main(int argc, char *argv[])
{
    /*  Initialize the image array  */
    float64 scien_data[LENGTH][HEIGHT][WIDTH];
    intn rank;
    int32 dims[MAXRANK], nt;

    /*  Get rank and dimensions of SDS  */
    DFSDgetdims("example.hdf", &rank, dims, MAXRANK);

    /*  Get number type of SDS  */
    DFSDgetNT(&nt);

    /*  Read the data array from the file if rank,  **
    **  dimensions and number type are correct      */
    if ((rank == MAXRANK) && (dims[0] == LENGTH) && (dims[1] == HEIGHT)
     && (dims[2] == WIDTH) && (nt == DFNT_FLOAT64))
        DFSDgetdata("example.hdf", rank, dims, scien_data);
}
```

2.4.6 Scientific Data Set Output (Current)

The programs in this section make use of the new SDS interface introduced with HDF 3.3. We recommend that people use this interface rather than the older interface. The program below writes a Scientific Data Set to the file “example.hdf”. This program takes no input.

```
#include "hdf.h"
#include "mfhdf.h"
#define LENGTH   3
#define HEIGHT   2
#define WIDTH    5
```

```
main(int argc, char *argv[])
{
    /*  Initialize the image array  */
    static float64 scien_data[LENGTH][HEIGHT][WIDTH] =
                { 1.,  2.,  3.,  4.,  5.,
                  6.,  7.,  8.,  9., 10.,
                 11., 12., 13., 14., 15.,
                 16., 17., 18., 19., 20.,
                 21., 22., 23., 24., 25.,
                 26., 27., 28., 29., 30. };
    int32 dims[3]         = {LENGTH, HEIGHT, WIDTH};
    int16 scale0[LENGTH]  = {2, 4, 6};
    int32 scale1[HEIGHT]  = {1234567, 2345678};
    float32 scale2[WIDTH] = {2.2, 4.4, 6.6, 8.8, 11.0};
    float64 avg           = 15.0;
    int32 start[3]        = {0, 0, 0};
    int32 fid, sdid, dimid0, dimid1, dimid2;

    /*  Open file and initialize SD interface  */
    fid = SDstart("example.hdf", DFACC_CREATE);

    /*  Create named data set  */
    sdid = SDcreate(fid, "Sample Data Set", DFNT_FLOAT64, 3, dims);

    /*  Set up dimension zero  */
    dimid0 = SDgetdimid(sdid, 0);
    SDsetdimname(dimid0, "Dimension 0");
    SDsetdimstrs(dimid0, "The zeroth dimension", "mm", "2d");
    SDsetdimscale(dimid0, LENGTH, DFNT_INT16, (VOIDP)scale0);

    /*  Set up dimension one  */
    dimid1 = SDgetdimid(sdid, 1);
    SDsetdimname(dimid1, "Dimension 1");
    SDsetdimstrs(dimid1, "The first dimension", "cm", "8d");
    SDsetdimscale(dimid1, HEIGHT, DFNT_INT32, (VOIDP)scale1);

    /*  Set up dimension two  */
    dimid2 = SDgetdimid(sdid, 2);
    SDsetdimname(dimid2, "Dimension 2");
    SDsetdimstrs(dimid2, "The second dimension", "m", "4.1f");
    SDsetdimscale(dimid2, WIDTH, DFNT_FLOAT32, (VOIDP)scale2);

    /*  Write the data array to the data set  */
    SDwritedata(sdid, start, NULL, dims, (char *)scien_data);

    /*  Add one local attribute  */
   SDsetattr(sdid, "Average", DFNT_FLOAT64, 1, (char *)&avg);

    /*  Add one global attribute  */
    SDsetattr(fid, "Date", DFNT_CHAR8, 9, "10/29/93");

    /*  Close the data set, file, and interface  */
    SDendaccess(sdid);
    SDend(fid);
}
```

2.4.7 Scientific Data Set Output (Current)

The program below reads a Scientific Data Set from the file "example.hdf" created by the program in the section above. This program produces no output.

```
#include <string.h>
#include "hdf.h"
#include "mfhdf.h"

#define MAXRANK   3
#define LENGTH    3
#define HEIGHT    2
#define WIDTH     5
#define DATESIZE  9

main(int argc, char *argv[])
{
    float64 scien_data[LENGTH][HEIGHT][WIDTH];
    int32 dims[MAXRANK];
    int16 scale0[LENGTH];
    int32 scale1[HEIGHT];
    float32 scale2[WIDTH];
    float64 avg;
    int32 start[MAXRANK] = {0, 0, 0};
    int32 fid, sdid, dimid;
    int32 i, index, rank, nattrs, ndatasets, nglobals;
    int32 nt, count, status;
    intn size;
    char name[80], date[80];

    /*  Open file and initialize SD interface  */
    fid = SDstart("example.hdf", DFACC_RDONLY);
    status = SDfileinfo(fid, &ndatasets, &nglobals);

    /*  Read global attribute  */
    if (nglobals == 1) {
        status = SDattrinfo(fid, 0, name, &nt, &size);
        if ((strcmp(name, "Date")) && (nt == DFNT_CHAR8) &&
            (size == DATESIZE))
            SDreadattr(fid, 0, date);
    }

    /*  Open first data set  */
    index = SDnametoindex(fid, "Sample Data Set");
    sdid = SDselect(fid, index);
    SDgetinfo(sdid, name, &rank, dims, &nt, &nattrs);

    /*  Read in data if everything looks okay  */
    if ((rank == MAXRANK) && (dims[0] == LENGTH) && (dims[1] == HEIGHT)
   && (dims[2] == WIDTH) && (nt == DFNT_FLOAT64))
   SDreaddata(sdid, start, NULL, dims, scien_data);

    /*  Read local attribute  */
    status = SDattrinfo(sdid, 0, name, &nt, &size);
    if ((strcmp(name, "Average")) && (nt == DFNT_FLOAT64) &&
        (size == 1))
        SDreadattr(sdid, 0, &avg);

    /*  Read dimensions  */
    dimid = SDgetdimid(sdid, 0);
    SDdiminfo(dimid, name, &count, &nt, &nattrs);
    if ((nt == DFNT_INT16) && (count == LENGTH))
        SDgetdimscale(dimid, scale0);

    dimid = SDgetdimid(sdid, 1);
    SDdiminfo(dimid, name, &count, &nt, &nattrs);
    if ((nt == DFNT_INT32) && (count == HEIGHT))
        SDgetdimscale(dimid, scale1);
```

```
    dimid = SDgetdimid(sdid, 2);
    SDdiminfo(dimid, name, &count, &nt, &nattrs);
    if ((nt == DFNT_FLOAT32) && (count == WIDTH))
        SDgetdimscale(dimid, scale2);

    /*  Close the data set, file, and interface  */
    SDendaccess(sdid);
    SDend(fid);
}
```

3. The Future: HDF-EOS

In this section we give a preview of our ideas about HDF-EOS. This discussion is not meant to be the final word, but is presented here for the purposes of generating feedback. The final design for HDF-EOS may differ in detail from what we present here.

3.1 Introduction: The Justification for HDF-EOS

We are in the process of establishing conventions for using HDF files to store EOS data. We plan to supply a subroutine library eventually that will encapsulate these conventions. We call the combination of the HDF standard and the EOS conventions the HDF-EOS standard, although strictly speaking it is not a new standard.

One reason for the existence of HDF-EOS, and a primary design goal, is to make EOS data as self-describing as possible. We need to be very clear what we mean by *self-describing*. Most of the fields in a self-describing data file are *computer-comprehensible*, in that a program would be able to read, say, the geolocation information and know that it *is* geolocation information. All of the information needed for this comprehension would be contained in the HDF-EOS standard, and would be encapsulated in the HDF-EOS subroutine library.

Another reason for the HDF-EOS standard is that there are common EOS datatypes that do not map directly to existing HDF datatypes. Examples would include gridded data, swath data, and point data. We therefore plan to create and support EOS specific datatypes in the HDF-EOS standard, in addition to the existing generic HDF datatypes.

There is one final philosophical point: one way to make a data file self-describing is to include in the HDF-EOS subroutine library a special routine for every single data product. In this case, every data product could be organized uniquely, with unique datatypes. The problem with this route is that one winds up with a standard and a subroutine library that becomes unnecessarily complex and difficult to maintain. We feel that a better route to self-description is by defining a small number of datatypes that will be used by all products.

Clearly, some EOS products are so unique that they will require special structures in the HDF-EOS file and special routines in the HDF-EOS library. One of our design goals is to keep those special structures and routines to a minimum.

3.2 Proposed HDF-EOS Datatypes

Our proposed data model for HDF-EOS is one where the HDF-EOS files contain a collection of *data objects*, each of a particular *datatype*. These data objects can be grouped together to create a hierarchy of objects. Each data object in the file must be of one of the HDF-EOS defined types (*datatypes*).

Some of the HDF-EOS datatypes map exactly to existing HDF datatypes. Some do *not* map to existing HDF datatypes, and are therefore constructed from a grouped collection of HDF datatypes. A design goal for the HDF-EOS standard is that there be no distinction to the user between HDF-EOS datatypes that are implemented with a single HDF datatype, and those that are implemented with multiple instances of HDF datatypes.

Note that the HDF-EOS datatypes described here are for the purposes of organizing standard data products only. Additional information that does not fit into this taxonomy (binary documents, for example), will be stored within the EOS system, but may not be delivered as components of a standard data product.

A brief summary of the proposed HDF-EOS datatypes is shown in Table 3-1. Each datatype is then discussed in more detail below.

Table 3-1. HDF-EOS Datatypes

Datatype	Definition	Implementation	SubTypes
ASCII Text	Descriptive Text	HDF Attribute	Plain, Formatted
P=V Metadata	Parameter=Value Info	HDF Attributes	Binary, Text
Science Data Table	Tabular Data	HDF Vdata	Standard, Index
Image	Raster Image	HDF RIS8,RIS24	8-bit Image, 24-bit Image
N-Dimensional Array	array of science data	HDF SDS	Array of Records, of Scalars
Grid	Data projected on grid	SDS with attributes	Rectangular, Structured Grid
Swath	Satellite ground track	multiple HDF SDSs	Simple, Complex
Point	Station Data	multiple Vdatas	Standard, Index
Data Dictionary	Dictionary of P=V words	HDF Attributes	
Structure	Group of Datatypes	HDF Vgroup	

3.2.1 ASCII Text

An ASCII text data object will contain text that is associated with the entire file, with groups of data objects, with individual data objects, or with a particular component of a data object. The association will usually be implicit, in that ASCII text data objects without any additional information would be associated with the file; those inside a structure would be associated with that structure, and so on. We plan to define two types of ASCII text data objects: Plain, and Formatted.

Plain — Straight ASCII text.

Formatted — Straight ASCII text that represents a particular format which must be processed such as PostScript, HyperText Markup Language (HTML), rich text format (RTF), and so on.

Our design goal for ASCII text is to support only those document types that are completely machine independent. This is why there are no provisions for storing binary text information, as there are no widely available machine independent binary text formats currently supported.

Note that our definition of 'support' is very limited: it means just that the ASCII text record will be flagged with the type of formatted text it contains. It will be the responsibility of the user (or user-developed software) to interpret the formatted text appropriately.

There is some debate about how to implement ASCII text data objects. The standard method would be to use HDF annotations, which have until recently been the only supported method for including text in HDF files. However, recent discussions suggest that we may want to consider using HDF attributes. One problem is that currently HDF attributes are limited to 32K bytes in size. This limitation will eventually be removed from the HDF standard.

HDF has always had attributes associated with its scientific dataset (SDS) data objects, but until recently the only supported attributes were numerical scales, calibration, names for the data and dimensions, fill values, and data range. With HDF Version 3.3, attributes have been generalized to include anything, including blocks of text.

3.2.2 P=V Metadata

The term 'Metadata' has come to mean many things within the ECS project. We always preface our use of the word with 'P=V', to make it very clear that we are talking about metadata (information about the data) that is of the form `parameter=value`. An example of P=V metadata is shown below in Figure 3-1.

```
Orbit number =                                  8957
Semi major axis =                               7229.1360 km
Eccentricity =                                  0.00117396
Orbit inclination =                             98.98316 deg
Right ascension of ascending node =             120.37838 deg
Argument of perigee =                           84.89226 deg
Mean anomaly =                                  84.89226 deg
Ascending - Descending flag =                   Ascending
Ascending - Descending node =                   34.71596536 deg
Time of ascending/descending node =             11:46:41:714
Julian day of ascending/descending node =       172
Year of asc/dec node =                          1990
Equator crossing node =                         34.71596536 deg
Equator crossing year =                         1990
Equator crossing Julian day =                   172
Equator crossing time =                         42401714 milisecs
State vector x pos component =                  -3658.3451 km
State vector y pos component =                  6240.8889 km
X velocity component =                          0.999273 km/sec
Y velocity component =                          0.586453 km/sec
Z velocity component =                          7.334024 km/sec
```

Figure 3-1. Example P=V Metadata

In a P=V data object, the parameters are field names such as '`Orbit number`', and the values are either single values or lists of values. Note that some values are numeric, some are text, some have units associated with them, and some formats are unique (time, for example).

The ECS project has defined two kinds of metadata: *Core* metadata that is required for all standard data products, and *product-specific* metadata that may be available for only some or even one data product. In either case, the exact format of the keywords and of the values will be fixed. This way, programs will be able to parse and comprehend the contents of the P=V metadata.

This enforcement of the P=V metadata keywords and values may be part of the HDF-EOS subroutine library, or it may be a separate component. The method that will be used for storing metadata in the ECS system has not yet been decided, and in any case is beyond the scope of this document. What is important, however, is the method used to store the P=V metadata inside the end-user HDF files. Two methods have been proposed: storing the text records inside a single ASCII text block as textual information, or storing each P=V metadata pair as a binary or ASCII format HDF attribute.

Using HDF attributes for P=V metadata is attractive from a performance point of view. It is easy to access the parameter names, and it is very easy to access the values associated with the parameters. In addition, it is trivial to support one-dimensional arrays as a set of values for a particular parameter when using HDF attributes.

There are several problems with this approach, however. Many if not most of the values are not strictly text or strictly numeric. As shown above, many values have units associated with them, or have unique formats. If we then store most values as individual text fields, this nullifies the main reason we went to attributes in the first place, namely that we would not have to convert ASCII numbers to binary numbers.

Other problems with using HDF attributes include the lack of any sort of hierarchical structure, the large number of data descriptors generated in the HDF "directory," and the lack of viewing of the P=V metadata with non EOS developed tools. In any case, we will study the issue and make a decision of which way to store P=V metadata in due course.

3.2.3 Science Data Tables

A science data table is easier to explain by example than via words. A very simple science data table data object is shown in Table 3-2. This is pretty much what most people think of intuitively when the word 'table' is mentioned.

Table 3-2. Example Table of Values

ID. No	Flux	Name
1	2.34	CygX1
2	-89.43	HerX1
3	0.0023	CygX3

4	1.115	Vela

More formally, a science data table consists of a series of named columns that are grouped together, such that all the entries from each column that are in the same row have a certain relationship. In the example above, there are three named columns, and each row represents a particular observation, with an **ID Number** associated with the observation, the radiation **Flux** from that source, and the **Name** of that source. Note that the columns can contain integers, floating point values, or text.

An HDF-EOS science data table will be implemented using the HDF Vdata interface. This interface has limitations that may or may not be important for HDF-EOS. For example, the number of bytes allocated to each item in a particular column is fixed, hence the byte size of each row is fixed. Another limitation is that the maximum size of each row is limited to 32K bytes. If it becomes important to do so, we may be able to eliminate one or both of these limitations by creating our own structures, but we do not yet have any plans to do so.

An additional limitation is that there currently is no easy way to associate attributes (such as units, etc.) with each column in the Vdata. We may wait for a future version of HDF to add this support, or we may consider adding some sort of support ourselves.

3.2.3.1 Types of Science Data Tables

We have designated two kinds of Science Data Tables:

Standard — Described above.

Indexed — A science data table where at least one column is an indexed pointer.

An indexed pointer is a proposed new component of HDF or HDF-EOS. It would point either to other data elements inside the HDF file using a standard HDF tag and reference number, or it might even point to a location within another data element inside the file.

An example of the first use may be for a table of images. The example science data table would then include information on each image such as a name, geographical location, and other relevant information. The indexed pointer column would contain the tags and reference numbers for images described in the other columns. A schematic of this example is shown in Figure 3-2.

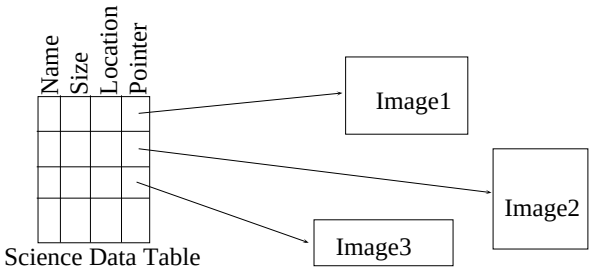

Figure 3-2. Example Indexed Science Data Table

An example of the second use may be for indexing into a larger table. Suppose for example you had a large list of events that are ordered by time and you wanted to see just the events for a particular day. To do this, you could do a binary search on the event list to find the first and last event of a day. But there is a better way.

You could instead construct a second list, an indexed science data table. There will be one entry in this list for each day. The list would have at least three columns: a column with the day value, a column for the number of events in that day, and an index pointer column that gives the record number within the event list table of the first event of that day.

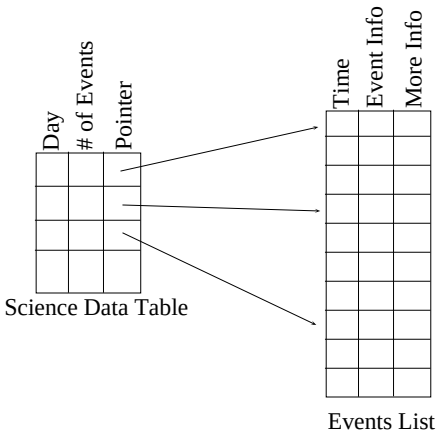

Figure 3-3. Second Example Indexed Science Data Table

We plan to support both uses of index pointers. Again, the issue is whether we wait for a future version of HDF to support index pointers, or whether we add the support ourselves as a part of the HDF-EOS standard.

3.2.4 Images

Images are data structures that are used solely to create pictures on computer screens or on printers. We will support images for the storage of browse packages only, *not* for the storage of scientific data. See the section below on using image objects to store data for a discussion of this issue.

3.2.4.1 8-Bit Images and 24-bit Images

We plan to support two types of browse images in HDF-EOS: 24-bit Raster Image, and 8-bit Raster Image. Both datatypes map exactly to corresponding HDF data objects, namely the RIS24 and RIS8 data objects. We do not foresee adding any EOS specific extensions to these image types.

24-bit Raster Image — Conceptually, it consists of three 2D arrays of values that represent a color image. Each array is exactly the same size, and represents the red, green, and blue values respectively of each pixel in a color raster image. The data values in each array are all eight bits in size. The term '24-bit raster' refers to the fact that there are 8+8+8=24 bits per pixel of color information.

For 24-bit raster images, it is possible to manipulate the *interleaving* of the red, green, and blue data values. There are three choices for the interleaving of 24-bit image data: by plane, by scan-line, and by pixel. If the image is *interleaved by plane*, then each red, green, and blue array is separate. If the image is *interleaved by scan-line*, it is stored as a sequence of red, green, and blue scan-lines. If an image is instead *interleaved by pixel*, then the image is considered as a single array of data values, where each data value is now a 24-bit grouping of red, green, and blue values for each pixel. For more information on interleaving, see section 3.3.1. The HDF documentation also has excellent discussions on the subject.

8-bit Raster Image — Consists of a single 2D array of values that represent a color image. Each value in the array does not, in itself, represent a color value, but instead is an index into a separate palette. Each entry in this palette represents a single color, with three values, one each for red, green, and blue. The data values in the raster array are eight bits in size, hence the term '8-bit raster'.

We show an example in figure 3-5 below of an 8-bit raster image. The **Data** array contains data values between 0 and 255 that represent the indices into the *palette*, or **Color Table**. Each entry in the palette contains values for red, green, and blue. These values are then finally used to display each pixel as a color in an **Image** on the screen or printer.

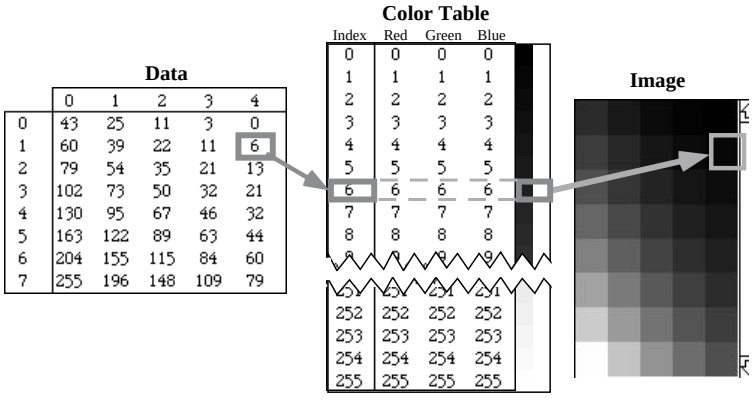

Figure 3-4. An 8-Bit Raster Image Example

8-bit images have some advantages over 24-bit images. One is that 8-bit images usually take much less disk space than 24-bit images, often a third as much. Also, 8-bit images give the user flexibility in how to display the image. By manipulating the palette, one can quickly change the

appearance of the image, such as bringing out distinctions between adjoining data values. The disadvantage of 8-bit images is that only 256 colors can be used to display the entire image, versus around 16 million colors (2^{24}) for 24-bit images.

Note by the way that a *grayscale* image is a special case of an 8-bit raster image, where the red, green, and blue values for each palette entry are equal, thus producing shades of gray. Note also that 8-bit raster images are also called *pseudocolor* images, since the color used for each pixel may or may not have any connection to a color we would see with our eyes when looking at the represented object.

3.2.4.2 Using Images to Store Data?

There is some confusion about using image data objects to store EOS data. Part of the confusion is that both image data objects and multi-dimensional array data objects can store 2D arrays of 8-bit data values. In addition, the processed output from many EOS instruments seems very 'image-like', almost like color or grayscale photographs.

There is, however, a vital distinction between image data objects and multi-dimensional array data objects. The former is meant to be used strictly for *display* purposes only. An image object contains all the information needed to generate a display.

To illustrate this, consider that there is only *one* correct way to display an image: the way specified in the object by the color planes, or by the color palette. Two images that have radically different data values inside the object but that produce the same display are considered identical. The internal structure and actual numerical values contained within an image object are considered irrelevant.

In contrast, a multi-dimensional array is meant to store *numbers* only. There is no right or wrong way to display a multi-dimensional array. There is no display information in the object at all. Two multi-dimensional arrays are considered identical only if the corresponding data values in each array match up exactly. The actual numerical values contained within the array object are considered extremely relevant.

We therefore propose that image data objects be used in the EOS system to store *browse images exclusively*. Browse images are meant to be visualizations of the data granules only, and are *not* meant to contain any scientifically valid data. They fit very well within the image data object paradigm.

In contrast, data from EOS instruments consists of nothing *but* scientifically valid data, by definition. These values should be stored in multi-dimensional arrays, or in one of the other structures discussed in this document. It should *not* be stored in an image data object, and we will not support such usage.

3.2.4.3 Image Compression

Both 8-bit and 24-bit image datatypes will allow compression. Two types are currently supported: Run length encoding, (RLE) and JPEG. RLE compression uses a very straightforward algorithm. It looks for strings of identical bytes and combines them into a single code. Although

RLE compression ratios tend to be modest (depending on the image), one big advantage of RLE is that it preserves all data values exactly. This is known as *lossless* compression, as opposed to *lossy* compression, where data values are not necessarily preserved.

JPEG compression is new to HDF, and is very complex. JPEG is a lossy compression technique. Indeed the goal of the JPEG algorithm is to maintain the appearance of the image with as little data as possible. The compression ratio is controlled by the selection of an image quality factor (called the Q-factor) when the image is written. The Q-factor can range from 0 to 100. The smaller the Q-factor selected, the more the image is compressed, and hence, the more potential image degradation.

JPEG compression is optimized for 24-bit images only. To compress an 8-bit image, it is first converted to a 24-bit image. This is not a complex or time-consuming task, but it does triple the image size before applying the compression algorithm. Because of this, compression ratios for 8-bit images with acceptable image degradation may not be that much better than RLE.

Since image data objects will be used only for browse image visualizations, and not for data, we plan to support both RLE and JPEG compression for 8-bit image data objects and JPEG for 24-bit image data objects.

3.2.5 Multi-dimensional Arrays

We expect multi-dimensional arrays to be one of the most widely used datatype in HDF-EOS. The general concept is simple: an N-dimensional array of values, all of which share the same data type, where N is between 1 and 32767. Our implementation of scalar multi-dimensional arrays maps exactly to the HDF scientific dataset data model (SDS), described in detail in Section 2 of this document and in the HDF documentation.

All arrays must be fully populated, either with real data values or with a fill value such as zero. Sparse arrays or ‘ragged’ arrays, where the size of each row or column changes, will not be supported directly.

3.2.5.1 Attributes and Dimension Scales

Associated with each array is a set of dimension scales, which consist of a series of numbers that mark the intervals of each dimension of the array. Also associated with each array is a set of *attributes*. Attribute examples include text labels for each dimension (for example, `Latitude`, `Longitude`, `Altitude`) and for the data (`Temperature`, for example), units for the dimensions and data, and calibration information.

Figure 3-5 shows an example of a two-dimensional array, with 3 by 3 = 9 data elements, dimensional scales for each axis (`0.5`, `1.0`, `1.5`), and text label attributes for each dimension (`X`, `Y`). Figure 2-5 also shows a two-dimensional array with dimension scales and text label attributes.

		X		
		0.5	1.0	1.5
Y	0.5	0.0350	0.4911	0.5744
	1.0	0.0714	0.2422	0.3305
	1.5	0.3853	0.9207	0.8485

Figure 3-5. A two-dimensional Multi-dimensional array with dimensions 3 by 3

Attributes are referred to in the form 'Parameter=Value' (such as '`Units=cm`', etc.), which is identical to the P=V Metadata paradigm discussed in the previous section. Should attributes and P=V metadata be one and the same thing? This is an issue we have not yet decided. In the HDF interface, attributes are implemented as a Vdata for every attribute.

For the purposes of the HDF-EOS Application Program Interface (API), we will almost certainly combine the concepts of P=V metadata and array attributes. How we implement them is much less clear. We may wind up supporting both text-based and binary Vdata-based forms of P=V metadata/attributes.

3.2.5.2 Scalar Arrays and Arrays of Records

We plan to support two kinds of multi-dimensional arrays: scalar arrays, and arrays of records.

Scalar Arrays — In a scalar array, every element in the array is a single number. The following number formats are currently supported: 32-bit and 64-bit floating point values, and 8-, 16-, and 32-bit signed and unsigned integers. Support for N-bit signed and unsigned integers will be added when HDF 4.0 is released. All data values within a particular array must be of exactly the same number format.

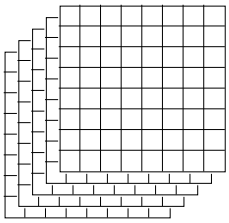

Figure 3-6. An 8 by 8 by 5 Scalar Array

Arrays of Records — In an array of records, every element in the array is a record consisting of many different numbers, each with possibly a different size and number type. Other than that, an array of records is just like a scalar array in that all data values within a particular array must be of exactly the same record format. In one sense, a record can be considered as just a very unusual number format.

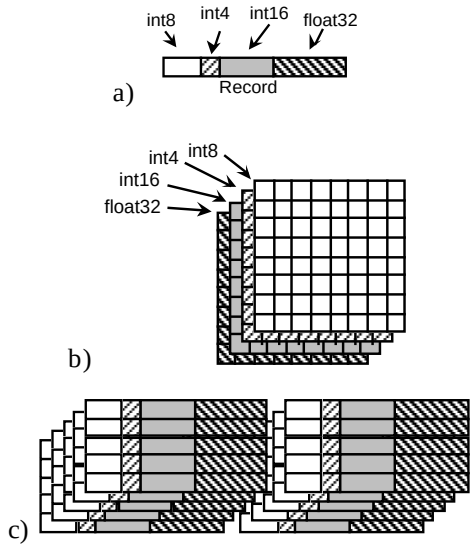

Figure 3-7. An Array of Records a) a single record b) a non-interleaved implementation c) an interleaved implementation

Consider as an example a 2 by 2 array of values, where at every location the data producer wants to store an integer, a floating point value, and a text string. Each record will be 11 bytes long: 2 bytes for the integer, 4 bytes for the floating point value, and 5 bytes for the text string. The disk layout for these records is shown in Table 3-3 below.

Table 3-3. Disk Layout of 2 by 2 Array of Records

	Byte Position in Record										
	1	2	3	4	5	6	7	8	9	10	11
Entry (1,1)	1		2.34				C	y	g	X	1
Entry (1,2)	2		-89.43				H	e	r	X	1
Entry (2,1)	3		0.0023				C	y	g	X	3
Entry (2,2)	4		1.115				V	e	l	a	
	integer		*floating point*				*text string*				

Table 3-3 is basically identical to Table 2-12, which is an example of a Vdata structure. This shows that there is yet another way to think of the array of records datatype, namely as a multi-dimensional Vdata structure.

The array of records structure is very useful for cases where many measurements or arrays of information are recorded in the same grid. For example, a user may want to record several spectral channels, latitude and longitude values, and so on at the same grid location.

3.2.5.3 Multi-dimensional Array Compression

At some point, compression of multi-dimensional arrays will be supported. We do not know exactly when, because we will wait until HDF officially supports it. We do not want to implement compression ourselves. If we did so, then programs compiled with the standard HDF libraries could not read array data from HDF-EOS files. We view this as unacceptable, and are willing to wait for the HDF implementation. Clearly however, we will only support *lossless* array compression techniques, since arrays will be used to store actual science data.

3.2.6 Grid Structures

Our grid structure is basically a multi-dimensional array, except that we guarantee that the first two or three of the dimensions are geolocation dimensions, such at latitude, longitude, and altitude. Additional dimensions could be used for non-spatial "locations" such as channel number, spectral range, and so on. In a grid structure, we will make it possible to specify a unique geolocation for every pixel, and to specify the pixel location for a given geolocation. We expect grid structures to be used extensively for Level 3 and Level 4 data.

We will support a wide variety of methods for specifying the geolocation information, and will detail a few of those methods here. But first we must describe the two different kinds of grid structures that we will support: *Rectangular*, and *Structured Grid*.

Rectangular — Data that has been projected and binned into a rectangular grid using a known methodology.

Data stored in a rectangular grid is directly displayable as if it were an image. This means that locations must have a well-formed spatial relationship not only to the locations to their left and right, but also above and below. A simple, well-formed relationship is true for *equal-angle grids*, where each row represents a particular latitude range and each column represents a particular longitude range.

A more complex, but still well-formed, relationship would hold for some projected arrays of data. For example, the data could be binned using the polar aspect of the Lambert Azimuthal Equal-Area projection into a 20 by 20 rectangular grid. Figure 3-8 approximates such a grid, showing a 30° latitude by 45° longitude graticule. This operation basically forms an image in the grid, but containing scientifically valid data values, as opposed to color values.

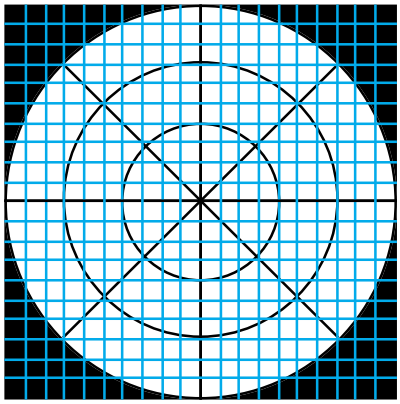

Figure 3-8. The Polar Aspect of the Lambert Azimuthal Equal-Area Projection Binned into a 20 by 20 Grid (an approximation)

Of course, metadata such as projection name, projection limits, and projection constants must be included in the data object in order to provide geolocation information and coverage of grid cells.

Structured Grid — Data that has been projected and binned into a non-rectilinear data structure using a known methodology.

Since a structured grid is conceptually non-rectangular, it is not directly displayable as if it were an image, but needs additional processing before display. This also means that there may not be a simple spatial relationship between adjoining locations in the grid data object.

An example of a structured grid would be a particular *equal-area grid* where every row representing a particular latitude has a different number of longitude bins than rows above or below it. An inexact representation of such a grid is shown in Figure 3-9. Although currently, equal-area grids are the only type of structured grids that we plan to support; we do want to keep open the possibility of supporting other non-rectangular grids.

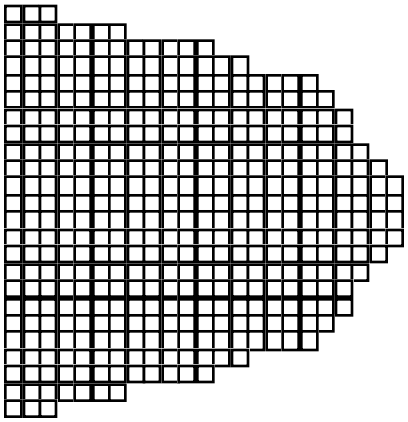

Figure 3-9. A Conceptual View of one Example of a Structured Grid

As in the case of the rectangular grid, metadata such as projection name, projection limits, and projection constants are included in the data object in order to provide geolocation information and coverage of grid cells

3.2.6.1 Geolocation Information

The whole point of a grid structure is to provide two functions: retrieving the geolocation of a particular array location (such as when clicking on an image representation of that data), and locating the data associated with a particular geolocation value (such as needed when overlaying an image representation with a vector overlay). All of our proposed methods for storing geolocation information must support both of these functions. We list some methods we plan to support in order of complexity. There may be other supported methods before we finalize the HDF-EOS standard.

Numerical Scales — For equal angle grids, the latitude, longitude, and altitude can be read directly from the X, Y, Z numerical scales of the array (see the write-up on multi-dimensional arrays above). Similarly, the array location for a particular geolocation can be calculated from a simple linear scaling. This method will generally hold for all rectangular grids where every pixel maps to a point on the globe (no "dead space") and both rows and columns can be charaterized by a single coordinate value.

Geolocation Arrays — Here the data array has two additional arrays associated with it: one that contains a latitude/longitude value for every location in the data array (‘Data Location to Lat/Long’ in the figure). The reverse function would be accomplished through a search of the Lat/Long array and interpolation, if necessary. This method of providing both functions is the most general, in that we make absolutely no assumptions about the arrangement of data in the data array. This method is also the most expensive in terms of disk space. Using this method, disk requirements may triple (or worse) for a given grid.

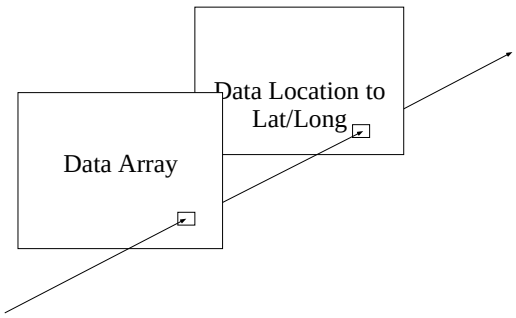

Figure 3-10. A data array with corresponding geolocation arrays

Projection Equations — The first method of providing geolocation information ('Numerical Scales') is both disk space and compute-time efficient, but it works only within very narrow restrictions. The second method of providing geolocation information ('Geolocation Arrays'), is completely general, but has a very serious disk space overhead problem.

Is there a method for providing geolocation information that it is intermediate between the first two? We believe the answer is yes. Two such intermediate methods are discussed below.

In the projection equations method, the *Geolocation to/from Data Location* functions can be provided using known mathematical transformations. We would then store the parameters for the transformations inside the grid structure. This method gives much more flexibility than the first method and, unlike the second method, requires almost no additional disk space.

One question is how to store the mathematical expressions. We could, for example, store the expressions as text strings that are parsed and executed when the grid structure is read or written. This method is very general, but would require reading and writing programs to have an expression interpreter built into the grid structure routines. We believe that the projection equations method results in unacceptable overhead and effort.

A second, much less powerful method would be to store a flag specifying the projection used and a block of scalar values needed by that particular projection. This method requires that the parameterized projection equations now be imbedded in the reading and writing subroutines. This method strongly encourages the selection of a small, controlled set of specifically supported projections and, when implemented, would serve to enforce that selection.

If there were hundreds of projections in common use for Earth science data, we would perhaps have to go with the expression parser option. However, we believe that the overwhelming majority of Grid structures will use fewer than 5 distinct projections. In that case, the overhead of including the projection equations in the reading code is minimal, and the loss of generality is not important. We are therefore leaning strongly toward the projection flag approach.

3.2.6.2 Structured Grids

An example of a structured grid is an equal-area grid where every row may have a different number of elements. We could consider making a general ragged-array data structure where we keep an index of each row size. However, we foresee that there will be only two or three types of structured grids that we will need to support. With such a small number, it would be more efficient to store just those parameters needed by the particular grids, without trying to be too general. The International Satellite Cloud Climatology Project (ISCCP) C Equal-area grid, for example, can derive the number of values for each row by reference to a few producer-selected parameters. It does not need a table to look up values for each individual row.

These structures will be so explicitly defined that it is not clear that a two dimensional array is the best storage mechanism. We could consider instead using a Vdata, or a one dimensional SDS to store the data values. The advantage of 1D arrays is that there is no wasted disk space. The disadvantages of 1D arrays is that they would be absolutely undisplayable as images with non-HDF-EOS tools (2D arrays would be viewable) and there is a small amount of computational overhead in figuring out the data location from row/column or lat/lon values. In addition, 1D SDS's and Vdatas have their own overhead considerations.

3.2.7 Swath Structures

A swath structure is similar to a grid structure, in that we guarantee geolocation information in both. One difference is that grid structures are meant to be used with Level 3 and Level 4 data, where one is abstracted somewhat from the actual instrument/platform combination, and swath structures are meant to be used with Level 1B and Level 2 data, where the data is more closely tied to the specifics of the instrument and platform.

A swath structure is meant to be used for data from instruments that generate a series of scans that are ordered along the orbital track. Typically, each of these scans cuts across the orbital track, as shown in figure 3.11(a). The same type of data structure can be used for data from a RADAR-like instrument, as shown in Figure 3.11(b).

In this example, the along-track motion of the platform causes the footprint to form a ribbon of M scans along the track, with each scan consisting of N locations. The data forms an array of observations of size M by N by L, where L is the number of data channels taken for each observation time. In the RADAR-like instrument, N is the number of beam angles in the sweep direction (azimuth), M is the number of angles in the elevation direction, and L is the number of range cells along each beam.

There are two major components to swath structures: the swath geolocation information, and the swath data arrays.

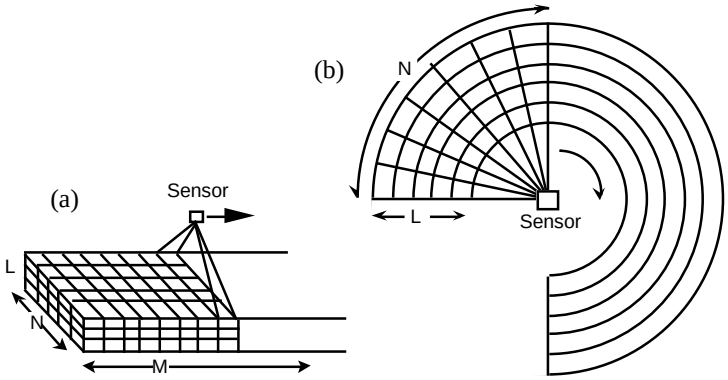

Figure 3-11. Two typical swath data scenarios a) a satellite swath b) a RADAR swath

3.2.7.1 Swath Geolocation Information

Our primary consideration in swath geolocation information is to be able to provide a latitude/longitude value for a particular element in a data array. This geolocation information would be used to display the data in appropriate projections on earth surface outlines.

In the grid structure discussion, we talked about three ways to specify geolocation: through the use of *numerical scales*, through *geolocation arrays*, and through *projection equations*. The first method can never work for swaths, since it depends on the data being arranged by rows of constant latitude and columns of constant longitude lining up on a grid. The second method, that of providing separate latitude/longitude arrays, does work for swath data. We expect this method to be the most widely used and supported.

It is not clear whether we will need to support the third method of providing mathematical transformations. If we do, the transforms will probably be specific to each instrument. An example would be where latitude/longitude was provided only for the first element in each scan, and a mathematical function was used to get the latitude/longitude for the rest of the elements in each scan line. This function would depend critically on the orbit of the platform, and the details of the scanning instrument.

No matter which method is used to store the geolocation information, the end result would either be a *real* latitude/longitude geolocation array, with at least as many elements as the largest data array in the swath, or a *virtual* geolocation array of the same size. For virtual arrays, many of the elements in the geolocation array may not actually exist in disk or memory, but are computed on the fly. The exact method of storage of the geolocation array will be designated in the metadata associated with the swath structure.

3.2.7.2 Swath Data Arrays

Typically, swath data arrays will be two dimensional arrays, corresponding to a 2D "image" of the ground along the orbital track. Sometimes, though, swath data arrays may be 1D arrays, where there is one element per scan (time, altitude, etc.). Additionally, swath data arrays could have 3 or more dimensions, where the additional dimensions are, say, channel number or altitude. In all cases, we will require that the first dimension of each data array (in row-major order) is the orbital track dimension (or elevation for RADAR-like instruments).

Note that we would not require that the size of the orbit track dimension be the same for every array. For example, in some instruments, information for one parameter is taken every scan, and for other parameters, information is taken every fourth scan. These orbital track 'skip' factors for particular arrays will be designated in the metadata associated with the swath structure.

We plan to designate swath structures with data arrays of varying resolution as *complex,* as opposed to *simple* swath structures where every data array is of the same size and resolution.

3.2.8. Point Structures

A point structure is exactly like a science data table except that we guarantee that the geolocation information is available and is in a standard form and location. We expect that this structure will be used primarily for station data. We have defined two kinds of point structures:

Simple — A science data table that consists of an ordered set of records containing various producer-defined data fields. A set of fields can be considered to give the location of the point in some coordinate system. For example, the location fields for a certain point data set might be latitude, longitude, altitude, and time. The remaining fields in each record are considered to be the data at the indicated location.

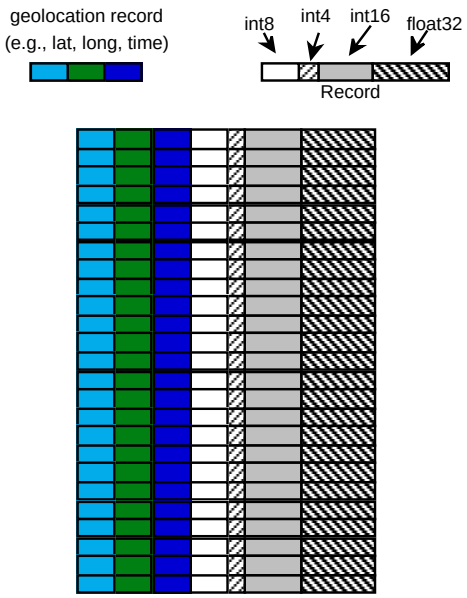

Figure 3-12. A Simple Point Data Structure

Indexed — An indexed point structure is exactly analogous to an indexed science data table, in that one of the fields is an index to another table or data object. For example, one could create one index table that contains information that stays the same from one measurement to the next, and another data table that contains information that changes between measurements.

For example consider a fleet of ships taking temperature data at various locations and depths. In this case, the *index* table might include such information on the ships such as a name, a mission code, and an index field point to the start of the data records for that ship in the *data* table. The data table might contain a field for the latitude, longitude, time, and each temperature measurement. Alternatively, the data table could contain pointers to the location of the appropriate ship entry in the index table.

3.2.9. Data Dictionary

To make sense of the metadata, one needs more information about each keyword. That is what the data dictionary can provide. Below is an example of a data dictionary for the orbit and attitude attributes.

```
Object = Orbit and Attitude
  Element
       Parameter Name =          Orbit number
       Definition =
       Format =                  9999
       Type =                    Integer
       Length =                  4
       Unit =
       Range =                   1 to 9999
```

```
     Comment =
Element
     Parameter Name =          Semi major axis
     Definition =
     Format =                  9999.9999
     Type =                    Float
     Length =                  9
     Unit =                    km
     Range =
     Comment =
Element
     Parameter Name =          Eccentricity
     Definition =
     Format =                  9.99999999
     Type =                          Float
     Length =                  10
     Unit =
     Range =
     Comment =
Element
     Parameter Name =          Orbit Inclination
     Definition =
     Format =                  99.99999
     Type =                          Float
     Length =                  8
     Unit =                          degree
     Range =
     Comment =
```

Figure 3-13. Example Data Dictionary

In addition to addressing P=V metadata, data dictionary entries will also be useful for fields used in defining point, gridded, and swath data products. The exact method of implementation of the data dictionary is still under investigation.

So should complete data dictionaries be stored inside every standard data product? This would create considerable duplication of information. On the other hand, it would make every single file pretty much self contained. This issue has not yet been decided, and will probably turn on the size of the data dictionary in relation to the size of the rest of the products.

3.2.10 Groupings of Data Objects

The *Collection of Structures* is the means by which data objects can be grouped within a file. The collection can include any defined data type, including other collections. See Figure 3-14 for an idea of how a collection of structures might be used. We plan to make liberal use of the collection of structures to provide organizational structure to data files. In fact, the Grid, Point, and Swath data types will each use the Collection of Structures as a wrapper to contain their constituent parts.

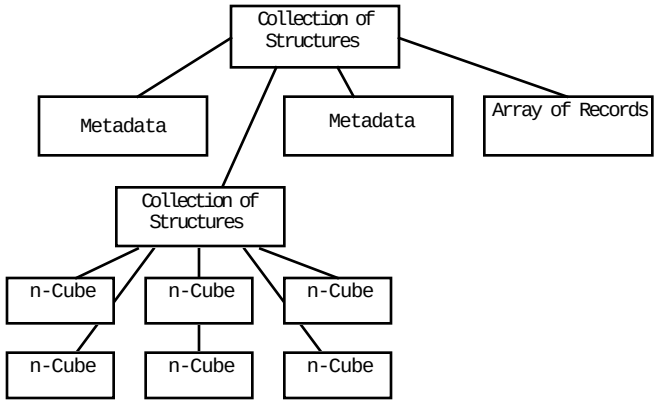

Figure 3-14. A Hierarchy of Collections

3.3 HDF-EOS Issues

In this section we discuss various issues involved with the HDF and HDF-EOS formats.

3.3.1 Data Interleaving

Data interleaving is an important subject from the perspective of efficiency of data access and, sometimes conversely, from the perspective of organizational simplicity. Efficiency is important at the lower processing levels, where data volumes are large and the number of users is small. Organizational simplicity is important at higher processing levels, where the needs of large numbers of users are a major concern. The interleaving method chosen by the data producer effectively dictates the best access method for the users, including further processing software and user visualization tools.

In HDF, interleaving is handled separately for each type of data object for which it is necessary. For instance, interleaving for SDSs (n-dimensional arrays of scalars) are handled differently from interlacing for RIS24s (24-bit raster images). The HDF library allows the user/producer to control the interleaving of SDSs, RIS24s, and Vdatas (tabular data). Note that in the HDF documentation; the term *interlacing* is used to refer to the concept of interleaving. The two terms will be used interchangeably in this document.

3.3.1.1 SDS Interleaving

The interleaving of an SDS is implied by the order of the dimensions as given to HDF in the creation of the SDS. The rule is that HDF will store the SDS on disk in the order specified by the dimensions given at creation time, using a row-major interpretation. For example, an SDS with dimensions defined as 2 by 3, or [2, 3] will be stored as an array of 2 rows of 3 columns each. Similarly, an SDS of dimensions [4, 5, 6] is regarded as an array of 4 planes of 5 rows of 6 columns.

C programmers will recognize this as ‘normal’ array order. FORTRAN programmers, however, should take note that this is the reverse order from the traditional column-major ordering used in FORTRAN programming. Given these rules for the specification of interleaving, a data producer can cause an SDS to be stored in any conceivable order, simply by re-arranging the order of the dimensions at creation time.

3.3.1.2 RIS24 Interleaving

The RIS24 interface will probably not be extensively used in the ECS, but it does provide flexible interleaving facilities, so we will discuss it here. In memory, a 24-bit raster image is implemented as an array of type *uint8* (unsigned char) with dimensions width by height by 3 (depth), in some order. As in the case of the SDS, the exact ordering of the dimensions is at the heart of the interleaving question.

By default, HDF assumes that you will be working with images interleaved by pixel. This means that an image that is 200 pixels wide by 100 high is equivalent to an array of bytes with dimensions [100, 200, 3] and that the red, green, and blue values that make up an individual pixel are stored contiguously.

You may also choose to interleave by scan-line or by scan-plane. Choosing interleaving by scan-line will require that you declare your array for the image mentioned above with dimensions [100, 3, 200], while scan-plane interleaving leads to dimensions of [3, 100, 200].

Regardless of the interleaving method used to *store* an image, you may request to *read* an image using any of the interleaving schemes. Of course, there is a performance penalty for any reorganization of the data.

3.3.1.3 Vdata Interleaving

There are two interleave options for Vdatas: FULL_INTERLACE and NO_INTERLACE. The terms are defined as follows:

FULL_INTERLACE — The first value from each field is collected into a record. Successive records contain subsequent values from each field. This can also be called record-oriented storage, since whole records are stored contiguously.

NO_INTERLACE — All data for the first field in the Vdata is stored contiguously in the file, then all the data for the second field, and so on. This can also be called field-oriented storage.

The method of interleaving is defined for a particular Vdata at creation time with a call to a special library function. If no interleaving method is specified, then FULL_INTERLACE is assumed. The interleaving method used is encoded in the Vdata.

As with the case of RIS24s, you can request that the library read Vdatas using either interleaving method. But again, you must pay a penalty in extra processing time to reorganize the data during the read operation.

3.3.2 Subsetting

The term “subsetting” has come to mean two very different processes in the context of the ECS system. The first meaning is where a user would like to extract a particular component from a standard data product granule. For example, she may want to look at just sensor channel 3. The corresponding data products include all channels in the same file, but she does not want to waste time downloading information she does not want. She would then request a subset of the standard data product, which delivers to her a file containing but that single array representing sensor channel 3. This sort of subsetting is easily handled by the existing HDF mechanisms.

The second type of subsetting is where a particular region of a data element is requested. For example, a user may want to specify a latitude/longitude subrange within which to provide data. In HDF, this situation will need to be handled differently for each type of data element. HDF provides the capability to read or write any subpart of any data element, but the logic required to decide which subpart is needed is highly dependent on the nature of the data element. Every effort will be made to provide intelligent subsetting of each datatype. Specifically, each of the geolocated datatypes *will* be capable of being subsetted by latitude/longitude box.

3.3.3 Subsampling

The term “subsampling” is used to refer to any number of algorithmic methods for spatially condensing a data set or data element into a smaller dataset. Two common techniques are:

Subsampling— Keeping only every n^{th} data value and throwing out the rest.

Spatial averaging — Combining spatially adjacent groups of n data values using a statistical averaging technique.

Each of these techniques can be applied somewhat independently on each spatial dimension of a data set. That is; the value of n can be different for each spatial dimension. Similar methods can be applied to non-spatial dimensions, as well (e.g., between multiple bands of data stored in the same datatype).

The base HDF library explicitly provides only the decimation technique. Furthermore, that technique is only provided for data stored in SDSs. We do not intend to provide additional subsampling techniques as part of the HDF-EOS library.

3.3.4 Browse Package Guidelines

The purpose of browse data in the ECS is to serve as a real-time on-line aid to ordering full data product granules. As such, its design must provide for fast access, small size, and simple organization that leads to easy interpretation. Our assumptions about these browse packages include the following:

- We expect that every Standard Data Product granule will have a corresponding browse package. This browse package can contain many different objects, as described below.
- We expect that instrument teams will generate these browse packages as part of their normal production processing.

- We expect that browse data packages will be static, not dynamic. That is; they will be generated once and archived rather than being produced in response to each user's request.
- We finally expect that some browse packages will consist of an "example" dataset plus additional data (e.g., cloud cover) rather than a subsampled product granule.

Objects that we plan to support in browse packages include the following (along with their associated metadata):

- Image _____8-bit raster image with palette, or 24-bit raster image.
- Overlay_____Raster image or vector table (special case of a Science Data Table). Used with Image.
- Table _____Science Data Table meant to be displayed as numbers.
- Plot _____Science Data Table meant to be displayed graphically.
- Text _____ASCII Text (Plain or Formatted).
- Animation_____A series of raster images.

We expect that the most common browse packages will consist of a single Image generated by a subsampling algorithm applied to a large array (along with associated metadata). Note that multidimensional array is *not* a supported browse package component. This prohibition will hopefully encourage the use of browse for visualizations of data, and not for delivering data itself.

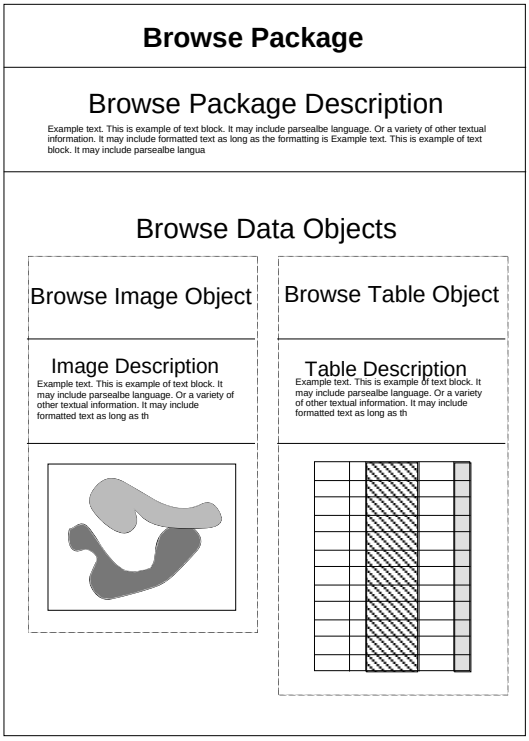

Figure 3-15. A Browse Data Package

We further propose the following general specifications for browse data products:

- A maximum overall package size of around one megabyte. A package much larger than this will tend to be less an interactive aid to ordering of data, and more of a data product in and of itself. Note that browse images can be compressed using lossy compression to help them fit under this limit (it can be lossy since all we care about is the image appearance, not the image data values). We further propose that no particular component of the browse package exceed the one megabyte limit.
- A target image size of around 620×620 pixels for images. Larger images will be allowed where absolutely required, if they can fit under the one megabyte limit when compressed.
- 8-bit raster images with an associated palette will be limited to say 150 entries in the palette. This limitation will reduce the chance that the image display will be substantively degraded when displayed on a system with a reduced color range. In addition, we would like fixed colors (colors used for burned in overlays, etc.) to be stored only in the highest and lowest palette entries (0 and 255, for example). This is because when a color palette is compressed to fit into a smaller color range, all entries except the first and last one may move by an entry or two.

3.3.5 File Sizes

HDF imposes a two gigabyte limit (2,147,483,648 bytes) on the size of a single, self-contained HDF file and on the individual physical files that make up a single logical HDF file. Some teams say that this limit is too low for their needs. However, we are primarily concerned with user access to data. A file approaching HDF's two gigabyte limit far exceeds the capabilities of the average current or near future science data user's computing facilities. We therefore do not consider the two gigabyte limit as unreasonable

We would even propose that user delivered HDF files not exceed around 512 megabytes in size. We further suggest that files be kept under 200 megabytes, wherever possible.

Note that NCSA has agreed to begin investigating the use of 64-bit addressing within HDF. A 64-bit address would increase the maximum HDF file size to the neighborhood of 10^{19} bytes (> 8,000,000 *tera*bytes). After the 64-bit address becomes standard in HDF and machines capable of handling such large files become widespread, the ECS project may, in consultation with the instrument data processing teams, re-evaluate the 512 megabyte file size limit.

3.3.6 Metadata and Data Dictionary Languages

There are two popular formats for storing 'P=V' metadata: Object Description Language (ODL) and Parameter Value Language (PVL). These two formats are very similar, in that PVL was derived from ODL. The Version 0 EOSDIS system used ODL for their metadata. However, PVL is more of an international standard. We plan to use the intersection of the PVL and ODL formats for our specification of P=V Metadata and Data Dictionary Data Elements. There has also been some discussion about using the Federal Geographic Data Committee (FGDC) metadata standard to define the spelling and the allowable values for the keywords themselves.

3.3.6.1 PVL Examples

An example PVL metadata group (ODL is similar) looks like the following:

```
/* Identifying information */
SPACECRAFT_ID=POLAR;
INSTRUMENT=PIXE;
START_TIME=1995-12-24T01:00:00;
FILE_TYPE=HDF;
```

Figure 3-16. A PVL Example

There are four keywords in this example, set to four specific values. The keywords cannot contain a space character, hence the use of the '_' character. Note that the values can be numeric, text strings, or in the case of the third parameter, time (which is in the PVL time format). Note also how every statement is terminated with an ';', and how comments are delimited by '/*'. '*/'.

These keywords can be grouped together, as the following example shows.

```
BEGIN_GROUP=UserProfile;
      UserType=EndUser;
      Name="Larry Q. Endal"
      Location="South Pole Station"
      PromptLevel=Novice  /* Other levels: Experienced, Expert */;
END_GROUP=UserProfile;
```

Figure 3-17. A Second PVL Example

Note the use of 'BEGIN_GROUP' and 'END_GROUP' to group together a series of keywords. Note also the use of double quotes to enclose longer text string values, especially those with embedded spaces.

Because of this grouping capability, PVL can also be used as a data dictionary language. In the following example, various parameters about two keywords are described.

```
/* Data dictionary example using PVL */
BEGIN_GROUP=ELEMENT_DEFINITION;
      NAME=SPACECRAFT_ID;
      DEFINITION="Spacecraft identifiers for science project"
      DATA_SYNTAX=Enumeration;
      DOMAIN_LIST={WIND,POLAR,GEOTAIL,CLUSTER,SOHO};
END_GROUP=ELEMENT_DEFINITION;

BEGIN_GROUP=ELEMENT_DEFINITION;
      NAME=START_TIME;
      DEFINITION="Start of coverage time for science project";
      DATA_SYNTAX=Date/Time;
END_GROUP=ELEMENT_DEFINITION;
```

Figure 3-18. A Third PVL Example

Some of these parameters, such as DEFINITION, are meant to be read by humans. Other parameters, such as DATA_SYNTAX, can be scanned by a computer program to decide how to read the values for a particular keyword.

3.3.6.2 Differences Between PVL and ODL

In general, PVL is a superset of ODL. There are however specific syntactical differences, which include:

- ODL expressions are always terminated by End-Of-Line (carriage return (CR), line feed (LF), or carriage return-line feed (CR-LF)). PVL expressions can also be terminated by End-Of-Line, but more typically are terminated by a semicolon (';').
- ODL comments always terminate with End-Of-Line. PVL comments terminate with '*/'.

If we specify that ECS metadata expressions always terminate with End-Of-Line, and that ECS metadata comments terminate with '*/' *and* with End-Of-Line, then our metadata should be parsable by both ODL and PVL readers. We view the compatibility with existing PVL and ODL software as a design goal for HDF-EOS Metadata.

3.3.7 Compression Methods

Compression algorithms come in two different types:

- *Lossless* — A compression method that allows the original data to be reconstructed exactly as it was before the compression algorithm was applied.
- *Lossy* — A compression method that can construct a reasonable facsimile of the original input data from the compressed data.

Lossless compression will be used for most EOS data, because the loss of perfectly valid information cannot be tolerated. The only exception so far noted is browse data; because browse data is only meant to be a *representation* of the data as an aid to ordering.

Currently, HDF only supports compression in the RIS8 and RIS24 interfaces. Since images are only to be used for browse purposes, data producers are free to choose either of the available compression methods: RLE (RIS8 only) and JPEG (RIS8 and RIS24). See the section on Images for more information.

NCSA is currently researching lossless compression techniques for use on SDSs. As these new techniques become available in the standard HDF library, the ECS team will evaluate each one and make further recommendations, as appropriate.

3.3.8 Performance Issues

3.3.8.1 Many Data Objects

During Pathfinder and Version 0 efforts (which used HDF 3.2), it was found that HDF files containing many data objects caused performance problems. The main symptom was extreme slowness in opening the file, although there was also a slowdown in accessing individual data elements.

The HDF 3.3 libraries have greatly improved the situation. The multifile SDS and Vdata interfaces dramatically reduce the number of times the file is opened and closed. These newer interfaces are capable of concurrently dealing with more than one open file, rather than covertly performing multiple file opens and closes like the older interfaces. The behavior of the newer interfaces effectively sidestepping the issue of slow file opens. This new library also implemented a more efficient, tree-based lookup scheme to locate data elements more quickly.

Although major improvements have been realized in access to files with many data objects, it is still wise to avoid creating files with more than about 500 individual elements. The HDF utility program 'hdfls' can be used to aid in counting the number of data objects in a file. When invoked with the '-l' option, hdfls will produce a list containing roughly one line for each data element in the file. The output of hdfls can be piped into the UNIX utility 'wc' or a similar program to count the number of data elements.

It is important to note that the number of data elements in an HDF file can greatly exceed the number of data objects, such as SDSs. For example, a single SDS can easily generate a dozen data elements that refer to its attributes, its numerical scales, and so on.

3.3.8.2 Large Data Objects

Many of the proposed data products for EOSDIS require the use of very large data elements. Specifically, individual Vdatas and SDSs for some products will likely reach tens of megabytes in size. When a data element reaches such a size, it becomes important to optimize I/O performance for that element.

One area where major performance gains can be realized is in data buffering. Buffering is where you fill up a memory buffer and then periodically write the buffer to disk. You optimally should have a large buffer, but not so large as to force the buffer to swap to disk, negating the point of having the buffer in the first place.

When doing buffering on HDF data elements, it is best to use the extended tag feature called a *linked-blocks* (see section 2.3.7), where the directory entries point not to the data, but to a list of blocks containing the data. HDF provides application programming level access to functions that can produce arbitrarily long chains of data blocks linked to form a large, expandable data element.

Each block that makes up a linked-block element is also a full-fledged (but somewhat hidden) data element. The implications here is that, if you write out an HDF file that contains only a single linked-block element and you ask the library to use a small block size, you may end up creating several thousand data elements, even though you only really wanted one. The key is to ask for a 'reasonable' block size.

Here are some general guidelines in choosing block sizes:

- One large block containing the entire data element yields the best performance. Of course, you will not always know how big that block should be, so it could be impractical to follow this rule in some cases.

- Choose a block size that makes a compromise between minimizing the number of blocks needed to write the expected nominal-size element (large block size) and minmizing the wasted space left in the last block, which may be only partially filled with actual data (small block size).

3.4 The HDF-EOS Library

It would be possible and relatively straightforward to read and write HDF-EOS files using existing HDF subroutines. We expect that several teams will actually do this to create their HDF-EOS files.

Clearly, however, this will create enormous duplication of effort. Every team, and many users, will wind up developing their own high level libraries for interfacing with HDF-EOS objects such as swath, grid, and point. In addition, the chance of accidentally violating HDF-EOS conventions when using generic HDF calls is fairly high.

We therefore propose developing an *HDF-EOS library* for reading, writing, and accessing HDF-EOS datafiles. Note that the material in this section is speculative, giving general direction only. None of the proposed routines should be taken seriously just yet.

3.4.1 Overview of the HDF-EOS Library

This library will include subroutines for reading and writing HDF-EOS structures, for verifying HDF-EOS structures, and for querying HDF-EOS structures (to obtain for example geolocation information, independent of the details of the structure).

The HDF-EOS library should be considered an extension of; and not a replacement for, the NCSA supplied HDF library. In fact, we plan to model the interface for the EOS specific datatypes on the new NCSA multi-file SDS API (Application Programming Interface). For example, to create a 3D SDS, a user may write the following code:

```
/*  Create named data set  */
sdid = SDcreate(fid, "Sample Data Set", DFNT_FLOAT64, 3, dims);

/*  Set up dimension zero (one and two would be similar):  */
/* name, text string, numerical array                      */
dimid0 = SDgetdimid(sdid, 0);
SDsetdimname(dimid0, "Dimension 0");
SDsetdimstrs(dimid0, "The zeroth dimension", "mm", "2d");
SDsetdimscale(dimid0, LENGTH, DFNT_INT16, (VOIDP)scale0);

/*  Write the data array to the data set  */
SDwritedata(sdid, start, NULL, dims, (char *)scien_data);

/*  Add one local attribute  */
SDsetattr(sdid, "Average", DFNT_FLOAT64, 1, (char *)&avg);

/*  Close the data set  */
SDendaccess(sdid);
```

Figure 3-19. Sample SDS Creation Code

Here the user created a 3D SDS, put information into each of the three dimensions (only one is shown here), wrote the data out, and set an attribute to the data.

The goal of this SDS API is to let the user treat the SDS as if it was a *single* entity (albeit with many components). In fact, a single SDS consists of *many* individual data elements (such as a vgroup, a few vdatas, three arrays, and so on), but user usually never needs to know that.

Our goal for HDF-EOS is to abstract the data structure in exactly the same way, but for HDF-EOS datatypes. For example, to write a swath structure a data producer might do something like:

```
/*  Create swath structure with 3 2D arrays                    */
sdid = EOSSwathcreate(fid, "Sample swath", DFNT_FLOAT64, 3, dims);

/*  Set up geolocation array  */
EOSSwathGeoType(dimid0, 1,"Time");  /*  1D Time array          */
EOSSwathGeoname(dimid0, "Scan");
EOSSwathGeostrs(dimid0, "Scan Dimension", "mm", "2d");
EOSSwathGeowrite(dimid0, LENGTH, DFNT_INT16, (VOIDP)scale0);

/*  Write one 2D data array to swath structure                 */
EOSSwathwrite(sdid, start, NULL, dims, (char *)scien_data);

/*  Add one metadata record                                    */
EOSSwathsetmet(sdid, (char *)&swathmeta);

/*  Close the swath                                            */
EOSSwathendaccess(sdid);
```

Figure 3-20. Possible HDF-EOS Library Usage

or something similar. Again, the idea is to abstract out the tens of separate HDF elements that make up the swath: the data producer sees a single structure that she writes to.

Another component of the HDF-EOS library will be query routines that will make it easy to get particular kinds of information out of these structures. For example, the user could query a particular pixel location of a gridded dataset and get the latitude/longitude value for that pixel, independent of the fact that there could be say 4 different ways that geolocation is actually stored in the file. For example:

```
/* Return lat/long of pixel location of grid object */
EOSGridPixtoLatLong(gid,pixelx,pixely,&lat,&long)
```

Figure 3-21. More Possible HDF-EOS Library Usage

3.4.2 HDF-EOS Library Organization

The HDF libraries are organized by interfaces, which map for the most part to HDF data objects. For example, the Annotations (DFAN) interface is used to read and write annotations, the Vdata (VS), Vdata Query (VSQ), and Vdata Field (VF) interfaces are used to read and write vdatas, and so on.

3.4.2.1 HDF-EOS Interfaces

We will also have interfaces in the HDF-EOS library. And like the HDF libraries, our interfaces will map directly to the HDF-EOS data objects. The list of HDF-EOS data objects from Table 3-1 is repeated below in Table 3-5.

Table 3-4. HDF-EOS Datatypes

Datatype	HDF Interface	Additional HDF-EOS Services
ASCII Text	Annotations (DFAN)	(none)
P=V Metadata	Attributes (SD)	(default), and check syntax
Science Data Table	Vdata (VS, VF)	(default)
Image	Raster (DFR8, DFP, DF24)	(default), plus add geolocation info
N-Dimensional Array	Scientific Dataset (SD)	(default)
Grid	Scientific Dataset (SD)	(default), plus add geolocation info
Swath	Scientific Dataset (SD)	(default), plus swath geolocation info
Point	Vdata (VS, VF)	(default), plus define geolocation info
Data Dictionary	Annotations (DFAN)	(none), besides a syntax check
Structure	Vgroup (V)	(default)

The first column in Table 3-5 lists the supported HDF-EOS data objects, the second column lists which HDF interfaces we plan to use to construct the HDF-EOS data objects. The third column lists the additional services that our proposed HDF-EOS library will provide to those HDF-EOS data objects, above and beyond what is supported by the interfaces listed in column 2.

Most HDF-EOS interfaces will include the creation of a Vgroup to group components; the addition of text and binary required metadata, and the ability to have people add their own metadata to these structures. This collection of services is what we mean by ‘(default)’ in this column.

3.4.2.2 The Organization of HDF-EOS Interfaces

Like the HDF libraries, the HDF-EOS interfaces will consist of routines grouped into collections. Most of the HDF-EOS interfaces will contain the following collections: Access routines, Read routines, write routines, verify routines, query routines, and metadata/attribute routines.

However, not all HDF-EOS interfaces will have all of these collections, and some HDF-EOS interfaces will contain additional, specialized collections. These collections are summarized in Table 3-6 below.

Table 3-5. HDF-EOS Interface Collections

Collection Name	Description
Access Object	Open access to object, start description of object
Read Components of Object	Select an object component, read it
Write Components of Object	Select an object component, write it
Verify Object	Make sure object is a valid HDF-EOS structure
Query Object	Get object information such as geolocation info
Metadata/Attribute Routines	Set, read, write associated metadata and attributes

3.4.3 HDF-EOS Library Issues

There are several features of HDF-EOS structures that will show up in many different interfaces. We therefore think it useful to discuss these common features in some detail.

Geolocation Structures— Computer readable geolocation is one of the primary driving considerations of HDF-EOS. We expect that there will be a small number of ways that geolocation is specified, within all geolocation structures (grid, swath, point). As described in the second on Grids, some of these methods may include latitude/longitude scales, 2D arrays of latitude/longitude (and possibly altitude), and mathematical transformations. Other methods may include lists of latitude/longitude values. Part of this standard will be to define the exact format the latitude/longitude values.

Pointer Structures—Both the science data table structures and the point structures have support for index pointers, in that one structure can refer to another structure, or a component of a structure. We need to establish conventions for these index pointers, so their use will be consistent across the file format.

Groupings—Every HDF-EOS structure will begin with a Vgroup, that will contain all the structures elements. There will be many things in common with each structure type, such as a class name specifying the structure type, a common place to put metadata, a common way to specify the organization of the structure, and so on.

Metadata/Attributes—Metadata and attributes will exist both at the file level, and at the individual data object level. The way that metadata is stored in both cases should be consistent. Our current thoughts are that any metadata specification would include a Vgroup to group all the metadata, an attribute with a large text record to store the ODL/PVL, text-based metatadata, and a series of attributes to store binary metadata. This binary metadata would include for the most part information needed directly by visualization and analysis programs, such as organizational information and geolocation information.

3.5 EOSView: An HDF-EOS 'Cracker Tool'

The HDF-EOS team is developing 'EOSView', a tool for examining and verifying HDF-EOS data files. To understand what features EOSView should have; we need to look at a typical user scenario.

3.5.1 EOS Visualization Needs

An EOS user first uses visualization to help in the *selection of data*. She next needs to make sure that the received data is what she wants. Visualization is used for the *verification of data*. Finally she wishes to perform analysis of the data, and present the results of the analysis to her colleagues. Here, visualization is needed for *analysis and presentation*. If developed with care, EOSView can address all three of these categories of needs.

Selection of Data—The EOS client will use EOSView to visualize browse images during the selection of data. Both programs will execute simultaneously on the user workstation, and will communicate with each other via scripts.

Verification of Data—For verification of data, EOSView must be a simple, easy to use application that works on all major platforms and with all types of EOS data.

Presentation and Analysis of Data—We need to make it easy to get EOS data in leading visualization and analysis systems. EOSView can provide file conversion routines in the program itself. And since EOSView can be driven by external scripts, these conversions can be initiated directly from the visualization systems (such as interface definition language (IDL)).

Table 3-6. Visualization Needs for EOS

Visualization Need	Examples	Implementation Plan
Selection of Data	Display Browse images, geolocated maps, etc.	Scripts generated by EOS client, sent to EOSView for execution
Verification of Data	Look at data file images, metadata, and auxiliary information	Interactive EOSView application
Analysis and Presentation of Data	Analyze data in depth, produce presentation slides	Translations available in EOSView, external scripting of EOSView from other application

When being used interactively, the basic paradigm of EOSView is the following. Every open data file is represented in the program by a single *Datafile Window*. Each Datafile window displays global information about a particular data file, and a listing of the contents of that data file. There can be many *Visualization windows* (generated by the visualization modules) associated with a single Data file window.

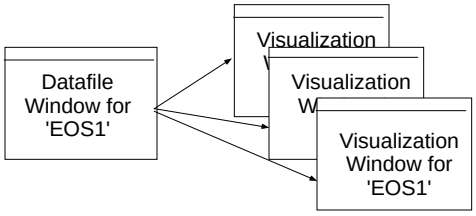

Figure 3-22. EOSView Windows

3.5.2 EOSView User Scenario

To help explain this design, we can follow a user through a typical session. The user first selects a file and clicks OK, which then prompts the program to display the contents of the file in the Datafile window.

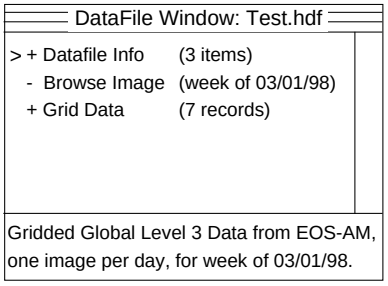

Figure 3-23. Initial Data file Window

The user now clicks on the *Datafile Info* line. When she does this, the GUI requests a list of the items contained in that group. Those items are then displayed, indented, under the group name.

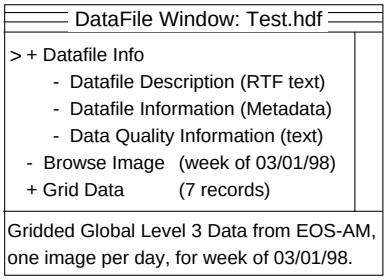

Figure 3-24. Data file Window with* Data file Info *group opened

The user next selects the line *Datafile Description (RTF text)*. This item is not a grouping, so instead the GUI asks for a display of that record. The program in turn makes a call to the appropriate visualization module, in this case the RTF display module. This module first displays the following dialog.

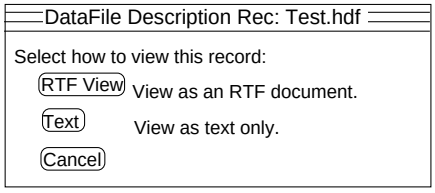

Figure 3-25. RTF View Dialog

If the user selects the first option, EOSView launches an external RTF viewer, and passes the contents of the record for display in that program. The relationship between EOSView and these external display programs is modeled after NCSA Mosaic, which uses external applications to play sounds and movies, and display images from Mosaic documents.

If the user selects the second option, then the GUI asks a visualization module in EOSView to display the data record as a simple text record. Next the user returns to the Data file window, and chooses to open up the *Grid Data* grouping. Seven Grid Data arrays are contained in the grouping (only two are shown here).

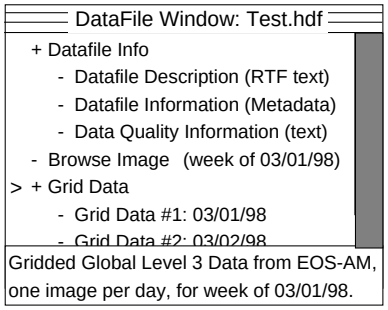

Figure 3-26. Data file Window with* Grid Data *group opened

Before getting deeply into these grid datasets, she decides that she wants an overview of the data file. She therefore clicks on the *Browse Image* record. The graphic user interface (GUI) then passes a command to a visualization module to display the browse image as a pseudocolor display, as shown below.

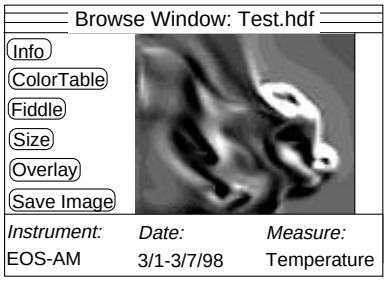

Figure 3-27. Pseudocolor display of browse image

In this case, EOSView knew of only one way to display that record. In the more general case, such as for *Grid Data* records, there may be multiple supported display methods (pseudocolor, contour, surface plot, arrays of numbers, and so on). In those cases, EOSView will ask the user for the method of choice, as was done for the RTF text case shown previously.

The pseudocolor browse image display window contains the image, annotations of the image (shown in the bottom of the window), and options that are specific to pseudocolor imaging. These options include seeing more information on the image, manipulating the pseudocolor color tables, fiddling with the colors, changing the size of the image, and adding overlays such as continent outlines and grid lines.

This page intentionally left blank.

Appendix A. Existing HDF Resources

A.1 Documents

As of the date of this document (December 1994), the current HDF version is 3.3 (release 4). Unfortunately, the documentation of this release is not complete or consistent. The current manuals available from NCSA are:

- *NCSA HDF Specifications, Version 3.2, UIUC, 1993.*
- *NCSA HDF Reference Manual, Version 3.3, UIUC, 1994.*
- *NCSA HDF Calling Interfaces and Utilities, Version 3.2, UIUC, 1993.*

Also relevant is the document developed for establishing HDF guidelines for Version 0 data:

- *EOSDIS Version 0 Data Product Implementation Guidelines , Draft Version 0, March 1994 (GSFC 50-003-04)*

A.2 User Support

For more information on HDF, contact:

NCSA Software Development Group, HDF
152 Computing Applications Building
605 East Springfield Avenue
Champaign, IL 61820
USA

Phone: 217-244-0072
E-mail: `hdfhelp@ncsa.uiuc.edu`
anonymous FTP: `ftp.ncsa.uiuc.edu`
URL: `http://hdf.ncsa.uiuc.edu:8001/`

This page intentioanlly left blank.

Appendix B. Obtaining, Installing and Using HDF

B.1 Obtaining the HDF Library and Documentation

The HDF source code distribution is available from NCSA via three methods:

B.1.1 FTP server

The Internet address of the FTP server is: `ftp.ncsa.uiuc.edu,` or `141.142.3.135`.

First, log in by entering anonymous for the name and enter your local e-mail address (`login@host.domain`) for the password. Next, change your directory to `HDF/HDF3.3rX/` (replace 'X' with the current release number).

Next, if you want *packed* source code, change your directory to `tar/` or `hqx/` or zip. Files in those directories need to be transferred using *binary* mode. If you want *unpacked* source code, change directory to `unpacked/` and transfer all the files in `unpacked/` and in its subdirectories to your host. For details, refer to the following file: `HDF/HDF3.3rX/unpacked/README.FIRST` (again, replace 'X' with the current release number).

B.1.2 Archive server

a. E-mail a request to `archive-server@ncsa.uiuc.edu`. Include in the subject or message line the word `help`, then press RETURN.

b. Send another e-mail request to `archive-server@ncsa.uiuc.edu`. Include in the subject or message line the word `index,` then press RETURN.

The information you receive from both the help and index commands will give you further instructions on obtaining NCSA software. Refer to Chapter one of "HDF Calling Interfaces and Utilities" for details.

B.1.3 US mail

A tape or CD-ROM archive of HDF is also available for purchase through the NCSA technical Resources Catalog. To obtain a catalog, contact:

NCSA Documentation Orders
152 Computing Applications Building
605 East Springfield Avenue
Champaign, IL 61820
USA

Phone: (217) 244-4130

B.2 Supported Platforms

As of the writing of this document, the NCSA supplied HDF libraries support the following hardware platforms:

Table C-1. NCSA Supported HDF Platforms

Manufacturer	Models	Operating Systems
Convex		UNIX
Cray	XMP,2	UNICOS
Thinking Machines	CM5	
Sun	3, 386, SPARC	SunOS
Silicon Graphics		Irix
DEC	DECstation 3100	Ultrix
DEC	VAX	VMS, Ultrix
DEC	Alpha	OSF/1
IBM	RT, RS6000	AIX
HP	HP9000	HPUX
NEXT		NextStep
Macintosh		MacOS
PCs		Windows, MS-DOS

B.3 Compiling HDF

Version 3.3 of the HDF library is compiled into a two-part library. The first part, `libdf.a` on UNIX systems, is referred to as the "base" library because it contains the low-level HDF I/O software layer and all the traditional pre-3.3 interfaces application program interface (APIs). The second part is called `libnetcdf.a` on UNIX systems and it contains HDF's new netCDF API and the new [recommended] SDS API. The second part of the library will be referred to as the netCDF/SDS portion.

B.3.1 The "Base" Library

The "base" library can be built with a single command from the `HDF3.3rX/hdf` directory where the subdirectories `src/`, `util/` and `test/` reside (again, replace 'X' with the current release number). The file `Makefile.template` is a generic, machine independent Makefile that you can modify if there is no Makefile already built for your machine.

For convenience, there are also machine customized makefiles. For example `MAKE.IBM6000` is a Makefile suitable for compiling HDF on an IBM RS/6000. Assuming you are on an IBM RS/6000, copy `MAKE.IBM6000` to `Makefile`:

```
cp MAKE.IBM6000 Makefile
```

and use the following commands to install different targets:

`make`

`make all` -- build the HDF library with the C and FORTRAN interfaces, the utilities, and C and FORTRAN test programs.

`make allnofortran` -- build the HDF library with only the C interfaces, the utilities and the C test programs.

Refer to the file `HDF3.3rX/hdf/INSTALL.TOP` (replace 'X' with the current version number) for more complete instructions on building the base HDF library.

B.3.2 The netCDF/SDS Portion

The new netCDF/SDS portion of the library can be installed by typing the following two commands:

```
configure
make
```

For more complete instructions on building the netCDF/SDS portion of the HDF library, please refer to the file `HDF3.3rX/mfhdf/README.HDF` (replace 'X' with the current version number).

B.4. Using the HDF Library

HDF programs may be written in either C or FORTRAN. Below are instructions on how to compile C and FORTRAN HDF programs on most UNIX systems.

B.4.1 Compiling C Programs

To use HDF routines in your C program, you must have the line

```
#include "hdf.h"
```

near the beginning of your code.

If you are on a Sun SPARC with the HDF include files in the directory `incdir` and the library files (`libdf.a` and `libnetcdf.a`) in the directory `libdir`, use the following command to compile a C program called `myprog.c` into an executable called `myprog`:

```
cc -DSUN -Iincdir myprog.c libdir/libnetcdf.a libdir/libdf.a -o myprog
```

or

```
cc -DSUN -Iincdir myprog.c -o myprog -L libdir -lnetcdf -ldf
```

Note that HDF supports ANSI C. If you have ANSI C but it is not the default C compiler on your machine, you will probably need to include the option `-ansi` in either command line shown above.

B.4.2 Compiling FORTRAN Programs

For FORTRAN programs, if your FORTRAN compiler accepts `include` statements, you may include `constant.i` and `dffunc.i` in your program. Otherwise, you need to use an editor to manually include any necessary declarations in your program. To compile a FORTRAN program `myprogf.f` on a Sun SPARC with the HDF include files in the directory `incdir` and the library files (`libdf.a` and `libnetcdf.a`) in the directory `libdir`, use:

```
f77 -o myprogf myprogf.f libdir/libnetcdf.a libdir/libdf.a
```

B.4.3 General Notes on Compiling

- If you do not require the functions supported in the netCDF/SDS portion of the library, the `libdir/libnetcdf.a` or `-lnetcdf` may be omitted from the appropriate command line shown above.
- On Silicon Graphics machines, you must add `-lsun` to the above command lines if you intend to use the netCDF/SDS portion of the library.
- For machines other than Suns, simply substitute the proper machine type for `SUN` in any of the command lines above. A list of the supported machine types appears in the file `HDF3.3rX/README` (replace 'X' with the current release number).

Appendix C. HDF-EOS Points of Contact

For specific HDF-EOS questions or HDF-EOS user support, send an email message to any of the following people:

bfortner@eos.hitc.com (Brand Fortner)

ted@ulabsgi.gsfc.nasa.gov (Ted Meyer)

dilg@ulabsgi.gsfc.nasa.gov (Doug Ilg)

If you would like respond by US mail, the mailing address is:

Brand Fortner
Applied Research Corporation
1616A McCormick Dr.
Landover, MD 20785
USA

For bug reporting related to HDF-EOS data products, send an email message to:

bfortner@eos.hitc.com (Brand Fortner)

dilg@ulabsgi.gsfc.nasa.gov

larry@eos.hitc.com (Larry Klein)

User contributions:

If a user or Distributed Active Archive Center (DAAC) would like to contribute source code or receive source code that was developed by other users or DAACs to generate HDF data products, contact any of the support personnel listed above.

Articles to *EOSDIS Processor*:

If you would like to send articles related HDF experiences and data products available in HDF for distribution, send it to Ted Meyer at the email address shown above.

This page intentionally left blank.

Glossary and Acronyms

8-bit Raster .....Refers to a Raster Image where each pixel is represented by a single byte. This allows each pixel to be displayed with one of only 256 possible colors (using an associated color table). Known in HDF as a 'RIS8'.

24-bit Raster .....Refers to a Raster Image where each pixel is represented by three bytes, one each for red, green, and blue. This allows each pixel to be displayed with one of over 16 million possible colors. Known in HDF as a 'RIS24'.

Annotation..... In HDF language, a plain text data element that can be used to describe or identify any other data element, an entire file, or a specific tag number.

API ..... Stands for Application Programming Interface. A set of functions designed for use by applications programmers. Often used interchangeably with the term 'subroutine library', or especially in this project, 'toolkit'.

Arrays of Records .....A proposed structure where every element in an array is not a number but a record. One could think of Vdatas as a one dimensional version of an array of records.

ASCII Text .....Plain textual data stored using the American Standard Code for Information Interchange.

Attribute ..... In HDF language (borrowed from netCDF), a text or binary data object used to store a single value or a list of values. Attributes can currently be associated with an SDS or an entire file, and are most often used for storing metadata.

Binning..... Describes the process of combining data taken at various locations and placing the information, properly interpolated, into bin locations that are defined on a particular grid.

Bitmap Image.....A synonym for *Raster Image*. Bitmap comes from the fact that every pixel location has associated with it a string of bits (usually 8 or 24).

Browse Package ..... In the EOS world, refers to a collection of images, tables, or text that is meant to be a *representation* of a data product. Browse packages are designed to be an *aid to ordering data*, and not to be data in itself.

CCSDS ..... Consultive Committee for Space Data Systems

CDF ..... Stands for Common Data Format. A standard data format developed at Goddard Space Flight Center.

Color Palette..... A synonym for *Color Table*.

Color Table.....A table used to map pixel values in raster images to actual colors. Color tables are usually 256 entries in size, corresponding to the 256 possible values in an 8-bit raster image. Each entry in the table consists of three numbers: a red, a green, and a blue value to uniquely specify the color for that pixel value.

Computer-Comprehensible In this document the phrase refers to fields in a datafile that a computer program can read and interpret appropriately. An example would be latitude/longitude values, where a program may use these values to display a correctly positioned map grid on an image.

Computer-Readable ..... In this document, computer-readable refers to fields that a computer program can read, but not necessarily interpret. An example would be where a program could display stored latitude/longitude values from a file, but not necessarily use those values in further calculations to say display a map grid.

CSDT ..... In the EOS world refers to Computer Science Data Type. Roughly corresponds to our HDF-EOS datatypes.

DAAC ..... Distributed Data Active Archive Center

Data Descriptor .....An internal 12-byte HDF structure containing a *Data Identifier*, an *Offset*, and a length that uniquely identifies and locates a data element within an HDF file. The HDF directory consists of a list of Data Descriptors (DDs).

Data Dictionary.....Refers to a record that contains detailed information about keywords; especially keywords used in metadata. An example would be an entry for 'Satellite_Name' that enumerates a keyword title ("Satellite Name"), a field type ("text"), a field width ("10 characters"), and perhaps allowed values ("EOS-AM, Tropical Rainfall Measuring Mission (TRMM), EOS-PM", etc.).

Data Element .....Refers to the individual components of *Data Objects* within an HDF file. For example, a Data Object of datatype 'scientific dataset' would consist of several data elements: one for each of the attributes, another for the array itself, and so on.

Data Granule.....In the EOS world, refers to a particular instance of a *Standard Data Product*.

Data Identifier.....In the HDF world, refers to combination of a *Tag* and a *Reference Number* that uniquely identifies a *Data Element* within an HDF file.

Data Location .....In this document, data location refers to values that are meant to be used to locate a particular data value. Examples of data locations would be X, Y, Z values, or Latitude, Longitude, Altitude values.

Data Model ..... A description of the conceptual data model of a particular scientific data format. For example, one netCDF data model is of a series of records, each a different time. Each record contains a series of n-dimensional arrays, each a different physical parameter. Contrast Data Model with *Disk Format*, which defines the actual physical organization of the disk files written with that scientific data format.

Data Object ..... A particular instance of a *Datatype.* For example, an HDF file may consist of a series of data objects, each of a different datatype (raster, n-dimensional array, and so on).

Data Product ..... Synonym for *Standard Data Product*.

Data Taxonomy ..... Just as the set of animals can be organized into groupings and subgroupings (birds, mammals, dogs, etc.), so can data. Our proposed method for grouping EOS data into HDF-EOS *Datatypes* is described in Section 3 in this document.

Datatype ..... Refers to classes of data structures such as *Grids*, *Swath* structures, *Science Data Tables*, and so on that will be supported in HDF-EOS. Often there will be an exact mapping of an HDF-EOS datatype to an HDF data object. However, there will be cases where an HDF-EOS datatype is made up of several HDF data objects grouped together.

DD Block ..... A physically contiguous group of *Data Descriptors* (DDs). One or more DD Blocks make up the DD List.

DD List ..... The DD List, which is made up of one or more linked *DD Blocks*, contains every DD entry in a HDF file. The DD List can be considered as the internal HDF file directory.

DD ..... See *Data Descriptor*.

Dimension Scales ..... Refers to the series of one dimensional arrays associated with a particular *Multidimensional Array* (SDS). These arrays, one per dimension of the SDS, list the data location values (latitude, longitude, altitude, for example) for each dimension in the array. Note that this way of describing the data locations for an array only works for regularly gridded data.

Disk Format ..... A description of the actual physical byte values and locations stored in a disk file written in a particular scientific data format. Contrast with *Data Model*, which refers to the conceptual and not physical organization.

ECS ..... Stands for EOSDIS Core System. Refers to the core software and hardware system used to support EOSDIS.

EOS ..... Stands for Earth Observing System.

EOSDIS ..... Stands for Earth Observing System Data and Information System. Refers to the ground based data archive and management system for EOS.

EOSView ..... Our multi-platform HDF-EOS analysis and visualization application. Also called an 'HDF-EOS cracker tool', for its ability to display HDF-EOS file contents and organization.

Equal-Angle Grid ..... A way of storing geolocated data where the size of each bin location is defined by fixed degree changes in latitude and longitude. The problem with this method of storing data is that each degree of longitude represents very different distances at high and low latitudes.

Equal-Area Grid .....A way of storing geolocated data where each the size of each bin location are defined by fixed distances. The problem with this method of storing data is that the number of bins has to be different for every latitude.

ESDIS..... Earth Science Data and Information System

ESDT.....Stands for Earth Science Data Type. Refers to a higher level structure of EOS data than CSDTs. For the most part, ESDT refers to particular *classes* of *Standard Data Products*: Every standard data product is of a particular ESDT.

Extended Tags .....An extended tag DD does *not* point directly to the data, but to a data element *defining where the data is and how it is stored*. This data object *may* point to the beginning of a linked list of data blocks that contain the entire data record. Alternatively, the extended tag record could define the data element as being stored in an *External Element* in another disk file.

External Elements .....An external element is an HDF data element that is stored not inside the physical HDF file, but as a separate physical file.

FGDC ..... Stands for Federal Geographic Data Committee. The FGDC has proposed a set of metadata data standards, for possible use in geolocated earth data. These standards include supported keywords, along with allowed values.

FITS ..... Stands for Flexible Image Transport System; a data format popular in the astronomy field.

ftp ..... file transfer protocol

Geolocation ..... Refers to data locations that are specific to physical locations on the Earth, or on another planetary body. Geolocation is usually (but not always) specified in terms of latitude, longitude, and altitude.

Granule..... Synonym for *Data Granule.*

Grid ..... Refers to a particular example of a *Gridding* scheme.

Grid Structure.....Refers to an HDF-EOS datatype that will be designed to support gridded data, by making the geolocation information for the grid data *computer-comprehensible*.

Gridding ..... For EOS, refers to schemes for dividing locations on the Earth or on a projection of the Earth into many bins or cells. Each bin has a unique spatial location on the Earth. Although independent of projection, gridding schemes should be chosen to map well to a given projection: for example, projections that favor one area of the Earth in some way should have many bins in that location.

GSFC..... Goddard Space Flight Center

GUI..... Stands for Graphical User Interface.

HDF..... Stands for Hierarchical Data Format. The format was developed and is maintained by NCSA at UIUC.

HDF-EOS..... A shorthand notation for our proposal of establishing EOS conventions for the organization of HDF files used by the EOS system and the software library that will implement and enforce them.

HTML ..... Hypertext Markup Language

http ..... hypertext transport protocol

IDL ..... Stands for Interactive Data Language. A cross-platform data manipulation and visualization tool developed by Research Systems, Incorporated.

Image..... Synonym for *Raster Image*.

Indexed Pointer .....A data value that by HDF-EOS convention refers to a particular data element, or a particular location within a data element. Can be used to create links between tables and data elements.

Interlacing ..... Refers to the process of deciding how to organize the storage of an array: in particular, deciding which data locations will be close to each other physically. For example, a 3D array that was interlaced by altitude would have every 2D altitude plane stored together.

Interleaving ..... Synonym for *Interlacing*.

ISCCP..... International Satellite Cloud Climatology Project

JPEG..... Stands for Joint Photographic Experts Group. A lossy compression method for 24-bit raster images. The method has been expanded to include 8-bit images, as well.

Length ..... In HDF, the number of bytes that comprise a data element.

Lookup Table ..... A synonym for *Color Table*.

Low-Level Interface ..... Refers to *APIs* that refer to data elements such as arrays, attributes, and so on. Compare to a High-Level Interface, where data is referred to as Swaths, Grids, and so on.

LUT ..... Look Up Table. Yet another synonym for *Color Table*.

Metadata ..... Data that describes data. This term is fairly nebulous, as one person's data is someone else's metadata. In this document, we use the term Metadata to refer to 'Parameter=Value' information that is associated with a Standard Data Product, such as "`SPACECRAFT_ID='EOS_AM' `".

Multidimensional Array .. A synonym for *N-dimensional Array*.

MultiFile SDS .....An *API* for *Scientific Datasets* that lets you access to more than one SDS in more than one file at the same time. Commonly known as the "SD" interface. The term can also be used to describe the data object produced by the API, which supersedes the *NDG* and the *SDG*.

N-Dimensional Array .....Refers to an array of any dimension that contains either scalar data values or a record of various data values at every data location in the array.

NASA ..... National Aeronautics and Space Administration

NCSA ..... Stands for National Center for Supercomputing Applications. HDF, NCSA Image, Datascope, NCSA Telnet, NCSA Mosaic, and Collage are all creations of NCSA.

NDG ..... Stands for Numeric Data Group. Refers to the data objects created with the 'DFSD' *SDS* interface. We recommend using the 'SD' interface, which produces *Multifile SDSs*, for EOS data.

netCDF ..... network Common Data Form. netCDF is another data format library, developed by Unidata, which is freely available and is primarily used by the atmospheric science community.

Numerical Scales..... Synonym for *Dimension Scales*.

ODL ..... Stands for Object Description Language. Developed by the Planetary Data System at the Jet Propulsion Laboratory, it is a text-based language for describing metadata and data dictionaries.

Offset..... In HDF, offsets are used to specify the location of data elements. These offsets are expressed as a number of bytes from the beginning of the file.

P=V Metadata .....Metadata that can be put in the form "P=V," where P is a parameter name and V is a single value or a list of values.

Palette..... Yet one more synonym for *Color Table*.

Point Data ..... Refers to data collected at random locations, that cannot easily be stored on a regular grid. An example would be a record of temperature measurements taken at various airports across the country.

Point Structure ..... Refers to a proposed HDF-EOS datatype for storing *Point Data.*

Projection ..... Used here to mean a set of transformation equations that map a sphere onto a flat surface.

Pseudocolor Image ..... Here, a synonym for *Raster Image*.

PVL ..... Stands for Parameter Value Language. Refers to a text-based language developed by CCSDS (Consultative Committee for Space Data Systems) for the storage of metadata and data dictionaries. It was derived in part from *ODL.*

Raster Image ..... A rectangular array that is meant to be displayed on a computer screen, with each element in the array corresponding to a particular pixel in the computer display.

Raster Image Group ..... In HDF, a structure for gathering the data elements required to represent a raster image. It is like a Vgroup, but is specific to raster images.

Record ..... In HDF, a term used to refer to the repeated sequence of fields in a Vdata.

Reference Number ..... A unique number assigned to an HDF data element to distinguish it from other element with the same tag number.

RIG ..... Abbreviation for *Raster Image Group.*

RIS24 ..... Abbreviation for the 24-bit *Raster Images* provided in HDF. Also refers to the subroutine library for reading and writing 24-bit raster images.

RIS8 ..... Abbreviation for 8-bit *Raster Images* provided in HDF. Also refers to the subroutine library for reading and writing 8-bit raster images

RLE ..... Stands for Run Length Encoding. A compression method used in the RIS8 interface wherein contiguous runs of pixels with the same color value are expressed as a single color entry with a pixel count.

RTF ..... Stands for Rich Text Format. An ASCII-encoded format for storage of formatted text.

Scalar Arrays ..... An N-dimensional rectilinear data structure in which all data values are of the same basic type (e.g., 4-byte integer).

Science Data Table ..... A proposed EOS datatype organized as a set of named columns and a set of rows where each row contains one entry for each column. Should be very similar to Vdatas.

Scientific Data Group ....An obsolete structure for gathering the data elements required to represent a Scientific Data Set. Also referred to as an SDG.

Scientific Dataset ..... A data model and API provided in HDF for the reading and writing of multidimensional homogeneous arrays of data and the attributes of such arrays. The term is also used to refer to the data object produced by the API.

SDF ..... Stands for Standard Data Format. Refers to the standard format used for EOS data. Currently the HDF file format has been designated as the SDF for EOS.

SDG..... See Scientific Data Group.

SDS ..... See Scientific Data Set.

Self-describing ..... With a self-describing data file, no outside information is needed to fully comprehend the contained data, other than a subroutine library encapsulating the file format design.

Single File Interface .....An API that allows access to only one file at a time. HDF has only recently begun to move away from this method of file access.

Standard Data Product ....

Station Data .....Term used to refer to data that comes from a fixed location, with respect to the Earth.

Swath..... A data model for the reading and writing of data oriented around satellite orbital track. An API for swath data will be designed for HDF-EOS.

Swath Structure ..... Our proposed EOS datatype for organizing swath data, especially with making sure geolocation is in standard places.

Table..... In this document, a synonym for *Science Data Table*.

Tag ..... In HDF-speak, a number assigned by the HDF library that identifies the nature or intended interpretation of the data in a data element.

TIFF ..... Stands for Tagged Image File Format.

TRMM ..... Tropical Rainfall Measuring Mission

UIUC..... Stands for the University of Illinois at Urbana-Champaign.

V0..... Usually refers to the operational prototype of the EOSDIS system.

V1..... Usually refers to the first release of the EOSDIS system.

Vdata ..... A record-oriented HDF data model and API provided in HDF. A Vdata corresponds to a data table, where each fixed-length record is made up of a set of individually named and typed fields which make up the columns of the table. *Science Data Tables* make extensive use of Vdata.

Vdata Field ..... A named and typed set of data values that make up one column in the table-like Vdata structure.

Vgroup ..... An arbitrary grouping mechanism in HDF used to signify associations between otherwise unrelated data objects.

Vset ..... An obsolete term used to refer to a certain combination of Vgroups and Vdatas used to store three dimensional polygonal data.

This page intentionally left blank.